

ООП.

Принципы проектирования.

Модификаторы доступа.

Геттеры, сеттеры, property.

Преподаватель: Тенигин Альберт Андреевич

- Beautiful is better than ugly.
 - Explicit is better than implicit.
 - Simple is better than complex.
 - Complex is better than complicated.
 - etc.
- Красивое лучше уродливого.
 - Явное лучше неявного.
 - Простое лучше сложного.
 - Сложное лучше запутанного.
 - И прочие

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one– and preferably only one –obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

DRY

Don't Repeat Yourself

Не повторяйся

KISS

Keep it stupid simple
Пусть оно будет простым до безобразия

(Keep it simple, stupid)

SOLID

- Single responsibility principle (принцип единственной ответственности).
- Open-closed principle (принцип открытости/закрытости).
- Liskov substitution principle (принцип подстановки Лисков).
- Interface segregation principle (принцип разделения интерфейса).
- Dependency inversion principle (принцип инверсии зависимостей).

S - Single responsibility principle

Каждый блок вашего кода должен выполнять одну задачу

```
class TicketAndCardChecker:
```

```
def check_ticket(ticket: str):  
    if ticket.isdigit() and len(ticket) == 10:  
        date = [int(ticket[i:i+2]) for i in range(0, 6, 2)]  
        try:  
            sale_date = datetime.date(2000 + date[2], date[1], date[0])  
            print(sale_date)  
        except:  
            return False  
    else:  
        if sale_date < datetime.date.today():  
            return True  
    return False
```

```
def check_card(card: str):  
    card = card.replace(' ', '')  
    if card.isdigit() and len(card) <= 17:  
        digits = list(map(int, list(card)))  
        for i in range(len(digits)):  
            if i % 2 == 0:  
                new_value = digits[i] * 2  
                digits[i] = new_value if new_value < 9 else sum(list(map(int, str(new_value))))  
        if sum(digits) % 10 == 0:  
            return True  
    return False
```




```
class TicketChecker:
```

```
    def check(ticket: str):  
        if ticket.isdigit() and len(ticket) == 10:  
            date = [int(ticket[i:i+2]) for i in range(0, 6, 2)]  
            try:  
                sale_date = datetime.date(2000 + date[2], date[1], date[0])  
                print(sale_date)  
            except:  
                return False  
            else:  
                if sale_date < datetime.date.today():  
                    return True  
        return False
```

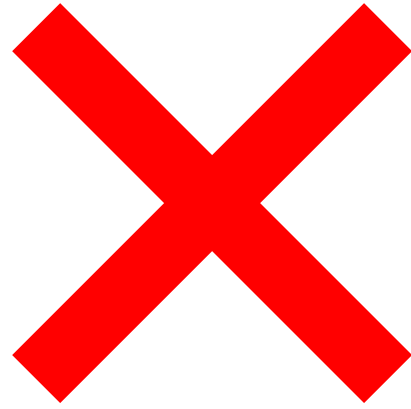
```
class CardChecker:
```


```
    def check(card: str):  
        card = card.replace(' ', '')  
        if card.isdigit() and len(card) <= 17:  
            digits = list(map(int, list(card)))  
            for i in range(len(digits)):  
                if i % 2 == 0:  
                    new_value = digits[i] * 2  
                    digits[i] = new_value if new_value < 9 else sum(list(map(int, str(new_value))))  
            if sum(digits) % 10 == 0:  
                return True  
        return False
```



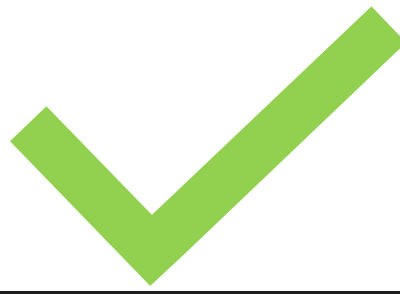
O - Open-closed principle

Ваши модули или библиотеки должны быть открыты для расширения, но закрыты для модификации.



 checkers.py > ...

```
1  class Building:
2      known_buildings = ['Arena', 'Lyceum', 'Park']
3
4      def valid_building(name):
5          return name in Building.known_buildings
```



```
checkers.py > ...  
1  from dotenv import load_dotenv  
2  from os import environ  
3  
4  class Building:  
5  
6      def __init__(self):  
7          load_dotenv()  
8          buildings = environ.get('BUILDINGS', default='')  
9          self.known_buildings = buildings.split()  
10         print(self.known_buildings)  
11  
12         def valid_building(self, name):  
13             return name in self.known_buildings
```

L - Liskov substitution principle

- Функции (и классы), которые используют указатели или ссылки на базовые классы, должны иметь возможность использовать подтипы базового типа, ничего не зная об их существовании.
- Подкласс не должен создавать новых мутаторов свойств базового класса.

```
class Email:

    def __init__(self, username: str, host: str):
        self.email = '{0}@{1}'.format(username, host)

    def isvalid(self):
        known_hosts = ['yandex.ru', 'gmail.com']
        for host in known_hosts:
            if self.email.endswith(host):
                return True
        return False
```

```
class SiriusEmail(Email):
```

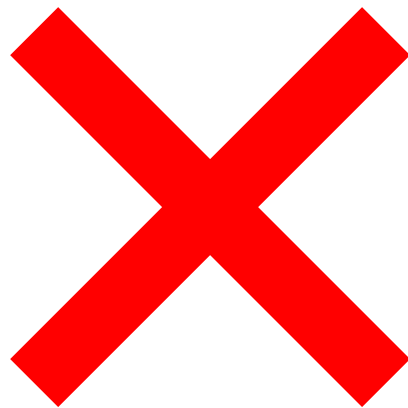
```
    def __init__(self, username: str):
        super().__init__(username, 'talantiuspeh.ru')
```



I - Interface segregation principle

Программные сущности не должны зависеть от методов, которые они не используют.

Отделяйте и разделяйте методы, не заставляйте пользователей (вашего кода) использовать ненужные или навязанные методы.



```
class Url:
```

```
    def opener(url: str, python_version: str, django_version: str):  
        if python_version.startswith('3.'):   
            if python_version < '3.8' and django_version < '4.0.1':  
                open_url(url)
```


D - Dependency inversion principle

- Модули верхних уровней не должны импортировать сущности из модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Модификаторы доступа в python

```
class Human:  
    def __init__(self, name):  
        self.name = name  
        self.age = 0  
        self.passport = Passport()
```

Искажение имен (name mangling)

```
class Human:

    def __init__(self, name):
        self.name = name # public
        self._age = 0    # private
        self.__passport = Passport() # hidden
```

```
class Human:

    def __init__(self, name):
        self.name = name # public
        self._age = 0     # private
        self.__passport = Passport() # hidden

petya = Human('Petya')
print(petya.name) # это нормально
print(petya._age) # это не разрешено, но вы можете это сделать
print(petya.__passport) # так сделать не получится
```

```
class Human:

    def __init__(self, name):
        self.name = name # public
        self._age = 0     # private
        self.__passport = Passport() # hidden

petya = Human('Petya')
print(petya.name) # это нормально
print(petya._age) # это не разрешено, но вы можете это сделать
print(petya._Human__passport) # не делайте так НИКОГДА
```

Геттеры и сеттеры

```
class Human:

    def __init__(self, name):
        self.name = name
        self.__age = 0    # hidden

    def get_age(self):
        return self.__age

    def set_age(self, new_age: int):
        self.__age = new_age
```

Python way

```
class Human:

    def __init__(self, name):
        self.name = name
        self.__age = 0    # hidden

    def get_age(self):
        return self.__age

    def set_age(self, new_age: int):
        self.__age = new_age

    def del_age(self):
        del self.__age

    age = property(get_age, set_age, del_age)
```

Python way (even more)

```
class Human:

    def __init__(self, name):
        self.name = name
        self.__age = 0    # hidden

    @property
    def age(self):
        return self.__age

    @age.setter
    def set_age(self, new_age: int):
        self.__age = new_age

    @age.deleter
    def del_age(self):
        del self.__age
```