Лабораторная работа №5
Использование возможностей объектно-ориентированного программирования в Python

Цель работы: закрепить применение базовой терминологии объектно-ориентированного программирования; научиться проектировать и разрабатывать классы, создавать программы с использованием классов и экземпляров классов/

Задание на лабораторную работу:

1. Изучить теоретические сведения.
2. Написать программу в соответствии с вариантом.

Теоретические сведения

В объектно-ориентированной модели языка Python существует две разновидности объектов: объекты классов и объекты экземпляров.

Переменные, принадлежащие экземпляру или классу, называют **полями**.

Переменные класса	*Переменные экземпляра*
принадлежат классу	принадлежат каждому отдельному экземпляру класса
доступ к ним могут получать все экземпляры этого класса.	у каждого экземпляра класса есть своя собственная копия поля
если любой из объектов изменяет переменную класса, это изменение отразится и во всех остальных экземплярах того же класса	изменение переменных объекта никак не связанно с другими такими же полями в других экземплярах

Функции принято называть **методами** класса.
Методы реализуют поведение, наследуемое объектами экземпляров.
Вызов метода экземпляра:

```
instance.method(args...)
```

автоматически преобразуется в вызов метода класса:

```
class.method(instance, args...)
```

где класс определяется в результате поиска имени метода по дереву наследования.
Фактически в языке Python обе формы вызова метода являются допустимыми.
Пример:

```
class NextClass: # Определение класса
     def printer(self, text): # Определение метода
           self.message = text
           print(self.message)
```

```
>>> x = NextClass() # Создать экземпляр
 # Вызвать метод экземпляра
 >>> x.printer('12345')
 12345
 >>> x.message
 '12345'
```

```
# Прямой вызов метода класса
>>> NextClass.printer(x, '12345')
12345
>>> x.message
'12345'
```

Всё вместе (поля и методы) принято называть **атрибутами** класса.

Атрибуты обычно присоединяются к *классам* с помощью инструкций присваивания внутри инструкции class, а не во вложенных инструкциях def, определяющих функции.

Атрибуты обычно присоединяются *к экземплярам* с помощью присваивания значений специальному аргументу с именем self, передаваемому функциям внутри классов. Переменная self связывается с объектом, к которому был применен метод класса,

и через эту переменную мы получаем доступ к атрибутам объекта. Когда этот же метод применяется к другому объекту, то self свяжется уже с этим другим объектом, и через эту переменную будут извлекаться только его свойства.

Основные отличительные характеристики классов в языке Python:

• Инструкция class создает объект класса и присваивает ему имя. Инструкции class обычно выполняются при первом импортировании содержащих их файлов.

• Операции присваивания внутри инструкции class (не вложенные в инструкции def) создают атрибуты класса (так же как и в модулях). После выполнения инструкции class атрибуты класса становятся доступны по их составным (полным) именам: object.name.

• Атрибуты объекта класса хранят информацию о состоянии и описывают поведение, которым обладают все экземпляры класса, – инструкции def, вложенные в инструкцию class, создают методы, которые обрабатывают экземпляры.

Отличительные характеристики экземпляров классов:

• Вызов объекта класса как функции создает новый объект экземпляра. Всякий раз, когда вызывается класс, создается и возвращается новый объект экземпляра. Экземпляры представляют собой конкретные элементы данных в вашей программе.

• Каждый объект экземпляра наследует атрибуты класса и приобретает свое собственное пространство имен(они первоначально пустые, но наследуют атрибуты классов, из которых были созданы).

• Методы класса получают в первом аргументе (с именем self в соответствии с соглашениями) ссылку на обрабатываемый объект экземпляра – присваивание атрибутам через ссылку self создает или изменяет данные экземпляра, а не класса.

Существует много методов, играющих специальную роль в классах Python.

Метод ***__init__*** известен как *конструктор*, так как он запускается автоматически на этапе конструирования экземпляра.

Если конструктор вызывается при создании объекта, то перед уничтожением объекта автоматически вызывается метод, называемый *деструктором*. В языке Python деструктор реализуется в виде предопределенного метода ***__del__***(). Следует заметить, что метод не будет вызван, если на экземпляр класса существует хотя бы одна ссылка. Впрочем, поскольку интерпретатор самостоятельно заботится об удалении объектов, использование деструктора в языке Python не имеет особого смысла.

Свойство ***__dict__*** - его значением является словарь, в котором ключи – это имена свойств экземпляра, а значения – текущие значения свойств.

```
class Example:
     n = 5
     def adder(self, v):
             return v + self.n
... # Свойство n и метод adder – это атрибуты объекта-класса
>>> w = Example()
>>> w.__dict__
{}          # у экземпляра класса сначала нет собственных атрибутов
>>> w.n = 8           #когда мы выполняем присваивание новому полю n экземпляра,
>>> w.__dict__        # у него появляется собственное свойство
{'n': 8}
```

Пример: использования переменной класса

```
class Robot:
    '''Представляет робота с именем.'''     #строка документации
класса
    population = 0          # Переменная класса, содержащая кол-во роботов
```

```
    def __init__(self, name):
        self.name = name   # Переменная self.name принадлежит объекту
        print('Инициализация {0}'.format(self.name))
        Robot.population += 1    # обращение к переменной класса:
                #переменная population увеличивается при «создании» робота

    def __del__(self):
        print('{0} уничтожается!'.format(self.name))
        Robot.population -= 1
        if Robot.population == 0:
            print('{0} был последним'.format(self.name))
        else:
            print('Осталось {0:d} работающих роботов'.format (Robot.population))

    def sayHi(self):
        print('Приветствую! Меня называют {0}'.format(self.name))

    def howMany():    # Выводит численность роботов.
        print('У нас {0:d} роботов'.format(Robot.population))

    howMany = staticmethod(howMany)
```

staticmethod — используется для создания метода, который ничего не знает о классе или экземпляре, через который он был вызван. Он просто получает переданные аргументы, без неявного первого аргумента
classmethod — это метод, который привязан к классу, а не к экземпляру класса.
Более подробно https://webdevblog.ru/obyasnenie-classmethod-i-staticmethod-v-python/

```
droid1 = Robot('R2-D2')
droid1.sayHi()
Robot.howMany()

droid2 = Robot('C-3PO')
droid2.sayHi()
Robot.howMany()

print("\nЗдесь роботы могут проделать какую-то работу.\n")
print("Роботы закончили свою работу. Давайте уничтожим их.")
del droid1
del droid2
Robot.howMany()
```

_____
Вывод программы:

```
Инициализация R2-D2
Приветствую! Меня называют  R2-D2
У нас 1 роботов
Инициализация C-3PO
Приветствую! Меня называют C-3PO
У нас 2 роботов

Здесь роботы могут проделать какую-то работу.

Роботы закончили свою работу. Давайте уничтожим их.
R2-D2 уничтожается!
Осталось 1 работающих роботов
C-3PO уничтожается!
C-3PO был последним
У нас 0 роботов
```

_____

!!!! Очень важно понимать **разницу между атрибутами объекта класса и атрибутами экземпляра класса**.

Пример:

```
class MyClass:
    x = 10                  # Атрибут объекта класса
    def __init__(self):
        self.y = 20         # Атрибут экземпляра класса

c1 = MyClass()              # Создаем экземпляр класса
c2 = MyClass()              # Создаем экземпляр класса
print(c1.x, c2.x)           # Выведет: 10 10
MyClass.x = 88              # Изменяем атрибут класса
print(c1.x, c2.x)           # Выведет: 88 88
print(c1.y, c2.y)           # Выведет: 20 20
c1.y = 88                   # Изменяем атрибут экземпляра класса
print(c1.y, c2.y)           # Выведет: 88 20
```

Пример: параметрам конструктора класса задаются **значения по умолчанию**

```
class Rectangle:
    def __init__(self, w = 0.5, h = 1):
        self.width = w
        self.height = h
    def square(self):
        return self.width * self.height

rec1 = Rectangle(5, 2)
rec2 = Rectangle()
rec3 = Rectangle(3)
rec4 = Rectangle(h = 4)
print(rec1.square())
print(rec2.square())
print(rec3.square())
print(rec4.square())
```

Основные идеи, лежащие в основе механизма перегрузки операторов:

• Имена методов, начинающиеся и заканчивающиеся двумя символами подчеркивания (__X__), имеют специальное назначение. Перегрузка операторов в языке Python реализуется за счет создания методов со специальными именами для перехватывания операций.

• Такие методы вызываются автоматически, когда экземпляр участвует во встроенных операциях (т.е. методы перегрузки операторов не надо вызывать по имени). Например, если объект экземпляра наследует метод __add__, этот метод будет вызываться всякий раз, когда объект будет появляться в операции сложения (+). Возвращаемое значение метода становится результатом соответствующей операции.

• Классы могут переопределять большинство встроенных операторов. Сюда входят операторы выражений, а также такие базовые операции, как вывод и создание объекта.

• Методы перегрузки операторов являются необязательными – если какой-то метод не реализован, это лишь означает, что соответствующая ему операция не поддерживается классом, а при попытке применить такую операцию возбуждается исключение.

• Некоторые встроенные операции, такие как вывод, имеют реализацию по умолчанию (в Python 3.0 они наследуются от класса object, являющегося суперклассом для всех объектов), но большинство операций будут вызывать исключение, если класс не предусматривает реализацию соответствующего метода.

Общие методы перегрузки операторов

Метод	Перегружает	Вызывается
`__init__`	Конструктор	При создании объекта: `X = Class(args)`
`__del__`	Деструктор	При уничтожении объекта
`__add__`	Оператор +	`X + Y`, `X += Y`, если отсутствует метод `__iadd__`
`__or__`	Оператор | (побитовое ИЛИ)	`X | Y`, `X |= Y`, если отсутствует метод `__ior__`
`__repr__`, `__str__`	Вывод, преобразование	`print(X)`, `repr(X)`, `str(X)`
`__call__`	Вызовы функции	`X(*args, **kargs)`
`__getattr__`	Обращение к атрибуту	`X.undefined`
`__setattr__`	Присваивание атрибуту	`X.any = value`
`__delattr__`	Удаление атрибута	`del X.any`
`__getattribute__`	Обращение к атрибуту	`X.any`
`__getitem__`	Доступ к элементу по индексу, извлечение среза, итерации	`X[key]`, `X[i:j]`, циклы `for` и другие конструкции итерации, при отсутствии метода `__iter__`
`__setitem__`	Присваивание элементу по индексу или срезу	`X[key] = value`, `X[i:j] = sequence`
`__delitem__`	Удаление элемента по индексу или среза	`del X[key]`, `del X[i:j]`
`__len__`	Длина	`len(X)`, проверка истинности, если отсутствует метод `__bool__`
`__bool__`	Проверка логического значения	`bool(X)`, проверка истинности (в версии 2.6 называется `__nonzero__`)
`__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__`, `__ne__`	Сравнивание	`X < Y`, `X > Y`, `X <= Y`, `X >= Y`, `X == Y`, `X != Y` (или `__cmp__`, но только в 2.6)
`__radd__`	Правосторонний оператор +	`Не_экземпляр + X`
`__iadd__`	Добавление (увеличение)	`X += Y` (в ином случае `__add__`)
`__iter__`, `__next__`	Итерационный контекст	`I=iter(X)`, `next(I)`; циклы `for`, оператор `in` (если не определен метод `__contains__`), все типы генераторов, `map(F, X)` и другие (в версии 2.6 метод `__next__` называется `next`))
`__contains__`	Проверка на вхождение	`item in X` (где `X` – любой итерируемый объект)
`__index__`	Целое число	`hex(X)`, `bin(X)`, `oct(X)`, `O[X]`, `O[X:]` (замещает методы `__oct__`, `__hex__` в Python 2)

Статические методы и методы класса

Внутри класса можно создать метод, который будет доступен без создания экземпляра класса (*статический метод*). Для этого перед определением метода внутри класса следует указать декоратор ***@staticmethod***.

Вызов статического метода без создания экземпляра класса:

```
<Название класса>.<Название метода>(<Параметры>)
```

Вызов статического метода через экземпляр класса:

```
<Экземпляр класса>.<Название метода>(<Параметры>)
```

```
class Foo:
   @staticmethod
   def add(x,y):
       return x + y
x = Foo.add(3,4) # вызов метода, x = 7
```

Пример:

```
class MyClass:
    @staticmethod
    def func1(x, y):                # Статический метод
        return x + y
    def func2(self, x, y):          # Обычный метод в классе
        return x + y
print(MyClass.func1(10, 20)) # Вызываем статический метод
c = MyClass()
print(c.func2(15, 6))        # Вызываем метод класса
print(c.func1(50, 12))       # Вызываем статический метод
                             # через экземпляр класса
```

!!!! Внутри статического метода нет доступа к атрибутам и методам экземпляра класса

!!!! В Python 3.0 не требуется объявлять метод, как статический, если он будет вызываться только через имя класса, но мы обязаны объявлять его статическим, если он может вызываться через экземпляр.

Методы класса – это методы, которые оперируют самим классом как объектом. Создаются с помощью декоратора ***@classmethod***. В качестве первого параметра в метод класса передается ссылка на класс (cls).

```
class MyClass:
    @classmethod
    def func(cls, x):           # метод класса
         print (cls, x)
MyClass.func(10)                     # Вызываем метод через название
класса
c = MyClass()
c.func(50)                   #   Вызываем   метод   класса   через
экземпляр
```

Можно использовать встроенные функции staticmethod и classmethod. Обе функции помечают объект функции как специальный, то есть как не требующий передачи экземпляра, в случае применения функции staticmethod, и как требующий передачи класса, в случае применения функции classmethod.

Пример:

```
class Methods:
    def AA(self, x): # Обычный метод экземпляра
       print(self, x)
    def BB(x):       # Статический метод: экземпляр не передается
       print(x)
    def CC(cls, x): # Метод класса: получает класс, но не экземпляр
```

```
        print(cls, x)
    BB = staticmethod(BB)   # Сделать BB статическим методом
    CC = classmethod(CC)    # Сделать CC методом класса.
```

Две последние операции присваивания в этом фрагменте просто переприсваивают имена методов BB и CC. Атрибуты создаются и изменяются с помощью операции присваивания в инструкции class, поэтому эти заключительные операции присваивания переопределяют инструкции def, выполненные ранее.

Фактически методы класса всегда получают ближайший класс в дереве наследования, поэтому:

• Применение статических методов, в которых явно указывается имя класса, может оказаться более удачным решением для обработки данных класса.

• Методы классов лучше подходят для обработки данных, которые могут отличаться для каждого конкретного класса в иерархии.

Основные идеи, лежащие в основе механизма **наследования** атрибутов:

• Суперклассы перечисляются в круглых скобках в заголовке инструкции class. Наследующий класс называется *подклассом* (производным классом), а наследуемый класс называется его *суперклассом* (базовым классом).

• Классы наследуют атрибуты своих суперклассов (интерпретатор автоматически отыскивает их, когда к ним выполняется обращение, если эти атрибуты отсутствуют в подклассах).

• Экземпляры наследуют атрибуты всех доступных классов. Каждый экземпляр наследует имена из своего класса, а также из всех его суперклассов. Во время поиска имен интерпретатор проверяет сначала экземпляр, потом его класс, а потом все суперклассы.

• Каждое обращение object.attribute вызывает новый независимый поиск. Интерпретатор выполняет отдельную процедуру поиска в дереве классов для каждого атрибута, который ему встречается в выражении запроса.

Поиск в дереве наследования выполняется снизу вверх – от экземпляров к классам и далее к суперклассам и останавливается, как только будет найдено первое вхождение искомого имени атрибута.

• Изменения в подклассах не затрагивают суперклассы. Замещение имен суперкласса в подклассах ниже в иерархии (в дереве классов) изменяет подклассы и тем самым изменяет унаследованное поведение.

Поскольку самые нижние определения в дереве наследования переопределяют те, что находятся выше, *механизм наследования* составляет основу специализации программного кода.

Пример: создать новый класс Class2, в котором будет реализован доступ ко всем атрибутам и методам класса Class1

```
class Class1:           # Базовый класс (суперкласс)
    def  func1(self):
        print ("Метод func1() класса Class1")
    def  func2(self):
        print ("Метод func2() класса Class1")

class Class2 (Class1): # Класс Class2 наследует класс Class1
    def  func3(self):
        print ("Метод func3() класса Class2")

c = Class2()            # Создаем экземпляр класса Class2
c.func3()               # Выведет: Метод func3() класса Class2
c.func1()               # Выведет: Метод func1() класса Class1
```

```
c.func2()                # Выведет: Метод func2() класса Class1
```

Пример: создать классы, которые адаптируют свой общий суперкласс (*замещение* унаследованных атрибутов, *предоставление* атрибутов, которые ожидается отыскать в суперклассах, и *расширение методов суперкласса*)

```
class Super:
    def method(self):
        print('in Super method')      # Поведение по умолчанию

    def delegate(self):
        self.action()                 # Ожидаемый метод

class Inheritor(Super):    #Наследует все методы суперкласса как есть
    pass

class Replacer(Super):     #Полностью замещает method суперкласса
своим
    def method(self):
        print('in Replacer.method')

class Extender(Super):           # Расширяет поведение метода method
    def method(self):
        print('starting Extender.method')
        Super.method(self)
        print('ending Extender.method')

class Provider(Super):           # Предоставляет необходимый метод
    def action(self):
        print('in Provider.action')

if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender): [1]
        print('\n' + klass.__name__ + '...')
        klass().method()
print('\nProvider...')
x = Provider()
x.delegate()[2]
```

Результат работы:

```
Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
```

---

[1] Программный код тестирования модуля в конце примера создает экземпляры трех разных классов в цикле for. Поскольку классы – это объекты, можно поместить их в кортеж и создавать экземпляры единообразным способом. Кроме того, классы (как и модули) имеют атрибут __name__ – он содержит строку с именем класса, указанным в заголовке инструкции class.

[2] При вызове x.delegate интерпретатор отыскивает метод delegate в классе Super, начиная поиск от экземпляра класса Provider и двигаясь вверх по дереву наследования. Экземпляр x передается методу в виде аргумента self, как обычно. Внутри метода Super.delegate выражение self.action приводит к запуску нового, независимого поиска в дереве наследования, начиная от экземпляра self и дальше вверх по дереву. Поскольку аргумент self ссылается на экземпляр класса Provider, метод action будет найден в подклассе Provider.

```
in Super.method
ending Extender.method

Provider...
in Provider.action
```

Хранение классов в модуле

Определения классов можно хранить в отдельном файле модуля, а затем импортировать этот модуль. Модуль и класс могут имеют одинаковые имена.
Например, пусть имеется следующий файл person.py:

```
class Person:
    ...
```

Чтобы получить доступ к классу, нам необходимо обратиться к модулю, как обычно:

```
import person         # Импортировать модуль
x = person.Person()   # Класс внутри модуля
```

Имена модулей начинаются со строчной буквы, а имена классов – с прописной.

Тестирование в процессе разработки

Создав класс, его необходимо протестировать: создать несколько экземпляров нашего класса и посмотреть содержимое их атрибутов, созданных конструктором. Такое тестирование можно выполнять в интерактивном режиме, но потребуется повторное импортирование модулей и ввод инструкций.

Для проведения более полного тестирования можно добавлять программный код в конец файла, содержащего тестируемые объекты. При этом лучше оформить тесты так, чтобы они выполнялись, только когда файл запускается как сценарий для тестирования, а не при его импортировании. Для этой цели можно использовать проверку атрибута __name__ модуля.

Пример: предусмотреть возможность импортировать файл и запускать его, как самостоятельный сценарий для самотестирования:

```
class New:
    def __init__(self, name):
       self.name = name

if __name__ == '__main__': # Только когда файл запускается для тестирования
    # реализация самотестирования
    bob = New('Bob Smith')
    sue = New('Sue Jones')
    print(bob.name, sue.name)
```

Теперь при импортировании файла интерпретатор создаст новый класс, но не будет использовать его. При запуске файла в качестве сценария интерпретатор создаст два экземпляра класса и выведет значения атрибутов для каждого из них.

Задания

Задание 1

Создать класс Students, предназначенный для обработки информации о студентах (фамилия, группа, стипендия).

Конструктор класса должен задавать значения для атрибутов – name, group, salary. Атрибут group по умолчанию должен быть «пзт». Атрибут salary по умолчанию 101.22.

Класс должен содержать метод вывода информации о студенте в виде:
ФИО: студент Иванов, группа пзт, стипендия 101.22

Класс должен содержать метод вывода стипендии для конкретного студента get_salary.

Класс должен содержать метод для вывода общей численности студентов в виде:
На данный момент всего студентов – 5

В программе должна быть предусмотрена возможность «поступления» (создать экземпляр класса) и «выпуска» студента (удалить экземпляр). При поступлении вывести на экран фразу «Здравствуйте, Иванов». При выпуске студента вывести на экран фразу «До свидания, Иванов».

Осуществите перегрузку арифметических операторов (для уменьшения и увеличения стипендии): __add__() – сложение, __sub__() – вычитание (как в примере):

```
a= Students ('Иванов', 'аэп', 120.10) # Здравствуйте, Иванов
b= Students ('Петров')                # Здравствуйте, Петров
print(a + 10)   #Иванов: нужно увеличить стипендию на 10
print(b - 10)   #Петров: нужно уменьшить стипендию на 10
```

Задание 2

B1-B6: Напишите игру по следующему описанию. Есть класс «НЛО» и класс «ЛюдиХ». У НЛО один объект, имеющий здоровье 1000hp. Создается от 1 до 5 людей Х (случайное число), имеющих здоровье 100 hp каждый. В случайном порядке НЛО и ЛюдиХ бьют друг друга. Тот, кто бьет, здоровья не теряет. У того, кого бьют, оно уменьшается на N очков от одного удара (N случайное число). После каждого удара надо выводить сообщение, кто кого атаковал, и сколько у противников осталось здоровья. Как только у НЛО заканчивается ресурс здоровья или погибают все ЛюдиХ, программа завершается сообщением о том, кто одержал победу.

B7-B13: Напишите игру по следующему описанию. В игре несколько уровней. Цель игры - победить драконов. На первом уровне – один дракон, на втором уровне – два, на третьем – три и т.д. Номер уровня вводит пользователь.

У каждого дракона устанавливается здоровье в 100 очков. У игрока – 500 очков. Пользователь вводит число – силу удара. На это число уменьшается здоровье дракона. Когда здоровье уменьшится до 0, дракон погибает, наступает битва со следующим (если в уровне несколько драконов). Дракон наносит удар – случайное число в диапазоне 10-90.

Игра завершена, когда все драконы уровня убиты или погиб игрок.

Задание 3

B1, B7: Создайте класс ПЕРСОНА с методом information(), позволяющим вывести на экран информацию о персоне, и с методом age(), служащим для определения возраста (в текущем году).

Создайте 2 дочерних класса:

- Студент (ФИО, дата рождения, факультет, курс),
- Преподаватель (ФИО, дата рождения, факультет, должность, стаж).

Каждый класс со своими методами вывода информации на экран и определения возраста.

Создайте список из N персон, выведите полную информацию обо всех на экран, а также организуйте поиск персон, чей возраст попадает в заданный диапазон.
Комментарий: В основной части программы организовать ввод количества персон. Текущий год можно определить с помощью:

```
from datetime import date
a = date.today()
a.year    # аналогично a.day, a.month
```

B2, B8: Создайте класс ТРАНСПОРТ с методом info(), позволяющим вывести на экран информацию о транспортном средстве, а также методом gruz() для определения грузоподъемности транспортного средства.
Создайте дочерние классы:

- Автомобиль (марка, максимальная скорость, грузоподъемность),
- Грузовик (марка, максимальная скорость, грузоподъемность, наличие прицепа, при этом если есть прицеп, то грузоподъемность увеличивается в два раза)

Каждый класс со своими методами вывода информации на экран и определения грузоподъемности.
Создайте список из N машин, выведите полную информацию на экран, а также организуйте поиск машин, удовлетворяющих требованиям грузоподъемности.

B3, B9: Создайте класс ДЕТСКИЕ ТОВАРЫ с методом info(), позволяющим вывести на экран информацию о товаре, и методом age(), позволяющим определить, предназначен ли товар для заданного возраста потребителя.
Создайте дочерние классы:

- Игрушка (название, цена, производитель, материал, возраст, на который рассчитана),
- Книга (название, автор, цена, издательство, возраст, на который рассчитана),

Каждый класс со своими методами вывода информации на экран и определения соответствия возрасту потребителя.
Создайте список из N товаров, выведите полную информацию из базы на экран, а также организуйте поиск товаров для потребителя в заданном возрастном диапазоне.

B4, B10: Создайте класс ПО с методом vyvod(), позволяющим вывести на экран информацию о программном обеспечении, а также с методом ex_date() для определения возможности использования (на текущую дату).
Создайте дочерние классы:

- Свободное (название, производитель),
- Коммерческое (название, производитель, цена, дата установки, срок использования)

Каждый класс со своими методами вывода информации на экран и определения возможности использования на текущую дату.
Создайте список из N видов программного обеспечения, выведите полную информацию на экран, а также организуйте поиск программного обеспечения, которое допустимо использовать на текущую дату.

B5, B11: Создайте класс ТРАНСФОРМЕРЫ с методом info(), позволяющим вывести на экран информацию о трансформере, и методом search() для поиска по любым критериям. Создайте дочерние классы:

- Автобот (имя, виды оружия, количество патронов, умеют бегать, стрелять, трансформируются в автомобили),

- Десептикон (имя, виды оружия, количество патронов, умеют бегать, стрелять, трансформируются в самолет)

Каждый класс со своими методами вывода информации на экран и поиска.
Создайте список из N трансформеров, выведите полную информацию на экран, а также организуйте поиск.

B6, B12, B13: Создайте класс НАПИТОК с методом info(), позволяющим вывести на экран информацию о напитках, и методом find() для поиска напитка по цене.
Создайте дочерние классы:

- Сок (название, цена, объем, срок годности, вкус),
- Вода (название, цена, объем, срок годности, с/без газа),

Создайте список из N записей, выведите полную информацию на экран, а также организуйте проверку срока годности по текущей дате.
Работу с датой можно организовать так:

```
from datetime import date
a = date.today()
a.year    # аналогично a.day, a.month
```

Задание 4

Реализовать игру «Угадай слово» с помощью ООП.

```
Попробуйте угадать слово по буквам. У вас есть 6 попыток.
Загаданное слово: _ _ _ _ _ _ _
Введите букву: p
Буква 'p' есть в слове!
Текущее слово: p _ _ _ _ _ _
Введите букву: y
Буква 'y' есть в слове!
Текущее слово: p y _ _ _ _ _
Введите букву: z
Буквы 'z' нет в слове. Осталось попыток: 5
Текущее слово: p y _ _ _ _ _
Введите букву: t
Буква 't' есть в слове!
Текущее слово: p y t _ _ _ _
Введите букву: h
Буква 'h' есть в слове!
Текущее слово: p y t h _ _ _
Введите букву: o
Буква 'o' есть в слове!
Текущее слово: p y t h o _ _
Введите букву: n
Поздравляем! Вы угадали слово: python
```

Создать класс Игра, содержащий методы:

- конструктор – задается случайное слово (может быть из списка), количество попыток
- метод для вывода текущего состояния слова с угаданными буквами
- метод для проверки есть ли введённая буква в слове
- в основном методе вызываются вышеперечисленные методы.

В основном блоке кода создается экземпляр класса и вызывается основной метод игры.

Контрольные вопросы:

1. Каково основное назначение ООП в языке Python?
2. В чем разница между объектом класса и объектом экземпляра?
3. В чем состоит особенность первого аргумента в методах классов?
4. Для чего служит метод __init__?
5. Как создать класс?
6. Как создать экземпляр класса?
7. Что произойдет, когда простая инструкция присваивания появится на верхнем уровне в инструкции class?
8. Как можно расширить унаследованный метод вместо полного его замещения?
9. Вызывается ли конструктор базового класса, если производный класс определяет собственный метод __init__()?
10. Что обозначает двойное подчеркивание в имени атрибута?
11. Как производится перегрузка операторов в классах на языке Python?
12. Приведите примеры перегрузки арифметических операторов.