*Тема 4.1 Введение в ООП* (**4+6=10час)**

***Лекционное занятие 1 (1 час)***

Основа ООП – совокупность объектов, относящихся к определенным классам. Класс описывает устройство объектов. Объект же - экземпляр класса.

История ООП начинается в 60-е года прошлого ХХ века. Одним из первых объектных языком можно назвать **Simula 67**. Он был разработан Кристен Найгаард и Оле-Джохан Даль в Норвегии с целью создания симуляторов. **Simula 67** включала в себя большую часть концепций объектно-ориентированного программирования: классы и объекты, подклассы (наследование), виртуальные функции, безопасные ссылки и механизмы, позволяющие внести в программу коллекцию программных структур, описанных общим заголовком класса (префиксные блоки). Алан Кэй из Xerox PARC использовал **Simula** как платформу для его разработки **Smalltalk** (первых версий языка в 1970-х), расширяя ООП с помощью интеграции пользовательского интерфейса и интерактивного исполнения.

В 80-х года ООП обретает популярность ввиду распространения С++, но общие принципы модели стали применяться во многих языках программирования.

*Можно отметить следующие преимущества ООП:*

* Простота. Предполагается, что программы, реализованные через ООП, легки для понимания, так как программа моделирует реальную концепцию мира.
* Высокая скорость разработки. ООП предполагает повторное использование классов, их параллельную разработку. Больше усилий прилагается к анализу и проектированию, снижая общие затраты на последующую разработку программ.
* Удобство в тестировании и обслуживании. Модульное представление кода позволяет выполнять обновление в случае ошибки без необходимости внесения объемных изменений. Программы легко расширяемы, так как новые функции и классы могут быть внедрены на основе существующих.

*Можно отметить и недостатки:*

* Создание программы требует полное представление о ее структуре.

Требуется определенный уровень подготовки со стороны программиста.

* При описании новых классов может неоправданно увеличиваться объем программы, ее сложность может расти в геометрической прогрессии.

Не каждый аспект при решении задачи реализуем через классы.

*В ООП существует три основных принципа построения классов:*

* Инкапсуляция – объединение в классе данных, и методов работы с ними, скрывая реализации от пользователя.
* Наследование – возможность создать новый класс на основе существующего, при этом все характеристики родителя присваиваются классу-потомку.
* Полиморфизм – возможность использования объектов с одним интерфейсом без информации о типе и внутренней структуре объекта.

Python соответствует принципам ООП. В Python всё является объектами – и строки, и списки, и словари, и всё остальное. Введем ключевые понятия:

* Класс — абстракция реального мира, специальный тип данных; класс описывает свойства и методы, которые могут быть доступны у подобных объектов;
* Объект (Object) (экземпляр класса, Class Instance) — частный случай класса.

В качестве примера можно привести следующий:

- Класс – ученик, объект – Петров Василий, конкретный ученик 8 А класса.

Класс содержит *поля* и *методы*. **Поля** ‑ *данные класса*, переменные, привязанные к нему. **Методы** – *действия класса*, которые можно выполнять над полями. Набор полей и методов определяет *интерфейс* для взаимодействия с классом.

Например, у класса "Ученик" полями могут быть *ФИО*, *Класс*, а методом – *Сгенерировать текст контрольной работы*.

Описание класса выглядит следующим образом:

**class** ClassName:

Элементы (атрибуты), реализованные в классе

В написании имени класса *принято* использовать ВерблюжийРегистр (название в несколько слов, записанных слитно, каждое – с заглавной буквы, формируя «горбатый» текст): MyCircle, MyPoint.

Объект создается вызовом класса через присваивание. Каждый раз, когда вызывается имя класса, создается новый экземпляр объекта этого класса.

Общая форма создания объекта экземпляра следующая: Obj = ClassName()

Создание полей класса похоже на создание переменной. Метод внутри класса создается так же, как и обычная функция, – с помощью конструкции **def**. Доступ к полям и методам осуществляется через указание объекта, используя точку:

Приведем пример описания класса "Точка". Данный класс имеет два поля - координаты точки.

**class** Point2D:

 x=50; y=30

p1=Point2D; **print**(p1.x,p1.y)

|  |  |
| --- | --- |
| Рис.1. Результат работы программы: |  |

Point2D - это название класса, x, y – поля класса. Эти поля доступны всем экземплярам класса. В данном примере p1 представляет собой экземпляр класса Point2D. Значения полей у экземпляра можно менять.

**class** Point2D:

 x=50; y=30

p1=Point2D(); p1.x=15; p1.y=-2; **print**(p1.x,p1.y)

Как было сказано ранее, описание метода схоже с описанием функции, но при описании метода используется аргумент – *self*. Он является общепринятым именем для ссылки на объект, в контексте которого вызывается метод. Этот параметр обязателен и отличает метод класса от обычной функции.

Вызов метода через экземпляр класса выглядит следующим образом:

<экземпляр класса>.<Имя метода>([<Параметры>])

Доступ к атрибутам и методам класса внутри метода производится через переменную *self*. При вызове метода не нужно передавать ссылку на экземпляр класса в качестве параметра, как это делается в определении метода внутри класса. Ссылку на экземпляр класса интерпретатор передает автоматически.

Добавим в описание класса Point2D метод для вывода:

**class** Point2D:

 x=50; y=30;

 **def** printP(*self*): **print**(*self*.x, *self*.y)

p1=Point2D(); p1.x=15; p1.y=-2; p1.printP()

|  |  |
| --- | --- |
| Рис.2. Результат работы программы: |  |

Расширим класс, добавив метод вычисления расстояния от центра до точки.

**class** Point2D:

 x=50; y=30

 **def** printP(*self*): **print**("x=", *self*.x,"y=", *self*.y)

 **def** distanceP(*self*): **return** (*self*.x\*\*2 + *self*.y\*\*2)\*\*0.5

p1=Point2D(); p1.x=15; p1.printP(); print(p1.distanceP())

|  |  |
| --- | --- |
| Рис.3. Результат работы программы: |  |

Атрибут может быть статическим и динамическим (уровня объекта класса). Для работы с динамическим атрибутом нужно создавать экземпляр класса.

В языке Python классы характеризуются следующими особенностями:

* в отдельности классы представляют собой определенные пространства имен;
* классы поддерживают создание набора объектов;
* пространства имен классов могут быть унаследованы;
* в классах можно реализовывать перегрузку функций (методов);
* в классах можно реализовывать перегрузку операторов.

***Практическое занятие 1 (1 час)***

1.Напишите программу для класса Вектор, заданного тремя координатами и именем. Создайте метод вычисления длины вектора.

Примерное решение задания может выглядеть следующим образом:

**class** Clvec:

 x=0; y=0; z=0; name="Vector"

 **def** depV(*self*): **return** (*self*.x\*\*2 + *self*.y\*\*2+*self*.z\*\*2)\*\*0.5

v1=Clvec(); v1.x=15; v1.y=-4; v1.z=0; v1.name="Вектор"; **print**(v1.depV())

2.Напишите программу для класса Телефон, заданного моделью и цветом.

Примерное решение задания может выглядеть следующим образом:

**class** Phone:

 color = 'Grey'; model = 'C385'

 **def** printP(*self*): **print**("модель - ", *self*.model, "цвет - ", *self*.color)

p=Phone(); p.printP()

3.Реализуйте класс Четырехугольник и метод поиска его периметра.

**class** Rec:

 x1=0; y1=0; x2=0; y2=0; x3=0; y3=0; x4=0; y4=0

 **def** perimetr(*self*):

 d1=((*self*.x1-*self*.x2)\*\*2+(*self*.y1-*self*.y2)\*\*2)\*\*0.5;

 d2=((*self*.x3-*self*.x2)\*\*2+(*self*.y3-*self*.y2)\*\*2)\*\*0.5;

 d3=((*self*.x1-*self*.x4)\*\*2+(*self*.y1-*self*.y4)\*\*2)\*\*0.5;

 d4=((*self*.x3-*self*.x4)\*\*2+(*self*.y3-*self*.y4)\*\*2)\*\*0.5;

 **return** d1+d2+d3+d4

4.Реализуйте класс Круг и методы вычисления площади и длины окружности.

**from** math **import** pi

**class** Circ:

 x1=0; y1=0; r=2

 **def** sqC(*self*): **return** pi\**self*.r\*\*2

 **def** lenC(*self*): **return** 2\*pi\**self*.r

c1=Circ(); c1.r=15; **print**(c1.sqC())

***Лекционное занятие 2 (1 час)***

Важную роль в ООП играет *конструктор*. В языке Python определяют конструктор класса как метод инициализации экземпляра класса.

Конструктор бывает двух типов: 1)Параметризованный; 2)Непараметрический

Определение конструктора стартует, когда создается объект класса. Конструкторы также проверяют, что у объекта достаточно ресурсов для выполнения любой задачи запуска.

В Python метод **\_\_init \_\_()** имитирует конструктор класса. Этот метод вызывается при создании экземпляра класса. Он принимает ключевое слово self в качестве первого аргумента, который позволяет получить доступ к атрибутам или методу класса. В конструктор можно передать любое количество аргументов во время создания объекта класса, в зависимости от определения **\_\_init \_\_()**. Обычно он используется для инициализации атрибутов класса. У каждого класса должен быть конструктор, даже если он просто полагается на конструктор по умолчанию.

Уточнение: В Python наличие знаков подчеркивания спереди и сзади в имени метода говорит о том, что он принадлежит к методам перегрузки операторов.

Приведем примеры описания и использования конструктора:

**class** Stud:

 **def** \_\_init\_\_(*self*, nm, id): *self*.id = id; *self*.nm = nm

 **def** display(self): **print**("ID: %d nName: %s" % (*self*.id, *self*.nm))

st1=Stud("Петров",1500); st2=Stud("Смирнов",1200); st1.display(); st2.display()

|  |  |
| --- | --- |
| Рис.4.Результат работы программы:  |  |

Это параметрический конструктор. Непараметрический используют, если мы не хотим инициировать значения в объекте, или если имеем только аргумент self.

**class** Stud:

 **def** \_\_init\_\_(*self*): **print**("This is non parametrized constructor")

 **def** show(*self*,nm): **print**("Hello",nm) #nm - не переменная класса!

st1 = Stud(); st1.show("John")

Конструктор без параметров используют для инициализации элементов класса.

**class** Stud:

 **def** \_\_init\_\_(*self*): *self*.id = 1300; *self*.nm = "Иванов"

 **def** display(*self*): **print**("ID: %d nName: %s" % (*self*.id, *self*.nm))

st1=Stud(); st1.display()

|  |  |
| --- | --- |
| Рис.5. Результат работы программы: |  |

В классе запрещены несколько конструкторов. Если несколько **\_\_init\_\_** написано в одном классе, то *последний из них перезаписывает все предыдущие* конструкторы, что отличает Python от многих других языков программирования.

Каждый класс Python хранит встроенные атрибуты, и предоставляет к ним доступ через оператор «.», как и любой другой атрибут:

**\_\_dict\_\_** — словарь, содержащий пространство имен класса.

**\_\_doc\_\_** — строка документации класса. None если, документация отсутствует.

**\_\_name\_\_** — имя класса.

**\_\_module\_\_** —имя модуля, где опреден класс (**\_\_main\_\_** в интерактивном режиме).

**\_\_bases\_\_** — могут быть пустые **tuple**, содержащие базовые классы, в порядке их появления в списке базового класса.

Например, при обращении: **print**(Stud.\_\_dict\_\_), Python выдаст следующее:



Рис.6. Результат работы программы

Встроенные функции класса описаны ниже в Таблице 1.

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| getattr(obj,name,default) | Используется для доступа к атрибуту объекта. |
| setattr(obj, name,value) | Используется для установки определенного значения для определенного атрибута объекта. |
| delattr (obj, name) | Необходима для удаления определенного атрибута. |
| hasattr (obj, name) | Возвращает True, если объект содержит атрибут. |

В классе также может быть реализован деструктор **\_\_del\_\_**(). Он вызывается перед уничтожением объекта. Этот метод может использоваться для очистки любых ресурсов памяти. Класс также можно описывать отдельным модулем. Затем импортировать его в основной модуль программы с помощью import SomeClass.

***Практическое занятие 2(1 час)***

1.Напишите программу для класса Треугольник, заданный тремя сторонами и реализуйте проверку вида (равносторонний, равнобедренный, разносторонний).

**class** Treug:

 **def** \_\_init\_\_(*self*, a, b,c): *self*.a = a; *self*.b = b; *self*.c=c

 **def** dispTr(*self*):

 **if** (*self*.a==*self*.b **and** *self*.b==*self*.c): **return** "равносторонний"

 **elif** (*self*.a==*self*.b) **or** (*self*.b==*self*.c) **or** (*self*.a==*self*.c): **return** "равнобедренный"

 **else**: **return** "разносторонний"

tr1=Treug(10,10,10); **print**(tr1.dispTr())

2.Напишите программу для класса Комплексное число, заданное действительной и мнимой частью. Реализуйте поиск абсолютного значения комплексного числа.

**class** Comp:

 **def** \_\_init\_\_(*self*, re1, im1): *self*.re = re1; *self*.im = im1

 **def** absC(*self*): **return** (*self*.re\*\*2+*self*.im\*\*2)\*\*0.5

c1=Comp(10,-5); **print**(c1.absC())

3.Напишите программу, описывающую класс Студент с атрибутами ФИО, оценки по математике и информатике и методом поиска среднего балла студента.

**class** Stud:

 **def** \_\_init\_\_(*self*, nm, mat,inf): *self*.nm = nm; *self*.mat=mat; *self*.inf=inf

 **def** dispB(self): **print**((*self*.mat+*self*.inf)/2)

st1=Stud("Петров",5,2); st2=Stud("Смирнов",4,4); st1.dispB(); st2.dispB()

*Обратите внимание, что метод не возвращает средний балл, а выводит его.*

4.Напишите программу, описывающую класс Книга с атрибутами название, количество страниц и цена с методом вывода на экран цены указанной книги.

**class** Book:

 **def** \_\_init\_\_(*self*, nm, page,pr): *self*.nm = nm; *self*.page=page; *self*.pr=pr

 **def** dispB(*self*, nnew):

 **if** *self*.nm==nnew: **return** *self*.pr

b1=Book("Стихи",500,350); **print**(b1.dispB("Стихи"))

***Лекционное занятие 3 (1 час)***

**Наследование** — концепция ООП, где можно объявить класс мало отличимый от существующего. Его называют дочерним, а прежний ‑ родительским. Здесь можно не просто создать дубликат, но и расширить его, то есть дочерний класс, наследуя атрибуты родителя, может переопределять их и добавлять свои, что очень полезно, поскольку повторно используется основная часть ранее написанного кода.

В языке Python может быть реализовано несколько видов наследования:

* Простое наследование

Синтаксис при описании дочернего класса выглядит следующим образом:

**class** DerivedClass(BaseClass):

 Тело дочернего класса DerivedClass, у которого родитель - BaseClass.

Рассмотрим пример (*работа декоратора рассмотрена в следующей лекции*).

**class** Person:

 **def** \_\_init\_\_(*self*, nm): *self*.\_\_nm = nm

 **@property** #декоратор явлеется методом «оберткой» для внутренних

 **def** nm(*self*): **return** *self*.\_\_nm

 **def** display\_info(*self*): **print**(f"Name: {*self*.\_\_nm} ")

**class** Student(Person):

 **def** educate(*self*): **print**(f"{*self*.nm} edducate")#здесь работает декоратор

st=Student("Петя"); st.display\_info()

Описан класс Person, который представляет *человека,* и далее от него наследуется класс Student, гдек Person добавляется метод educate().

Можно обращаться к унаследованным атрибутам: st.display\_info()

Обратите внимание, что для Student недоступны закрытые атрибуты типа \_\_nm. Например, нельзя в методе educate обратиться к приватному атрибуту *self*.\_\_nm:

**def** educate(*self*): **print**(f"{*self*.\_\_nm} edducate") #*запуск метода вызовет ошибку*.

* Множественное наследование

В языке Python осуществляется поддержка множественного наследования, то есть один класс можно унаследовать от нескольких классов:

|  |  |  |
| --- | --- | --- |
| **class** A: #класс A | **class** B: #класс B | **class** C(A,B): #C наследуем от A и B |

Можно использовать функции issubclass() или isinstance() для проверки отношений двух классов и объектоа.

* issubclass(sub, sup) возвращает **True**, если sub является подклассом sup.
* isinstance(obj, Class) возвратит **True**, если obj объект или подкласс класса Class.

При наследовании можно переопределить методы родительского класса:

**class** Parent:

 **def** my\_method(*self*): **print**('Вызов родительского метода')

**class** Child(Parent):

 **def** my\_method(*self*): **print**('Вызов метода наследника')

p=Parent(); p.my\_method(); c = Child(); c.my\_method()

|  |  |
| --- | --- |
| Рис. 7. Результат работы программы: |  |

Иногда возникает необходимость вызвать перекрытый метод базового класса в экземпляре производного класса. Например, при инициализации объекта. При создании объекта нужно убедиться, что будут вызваны метод \_\_init\_\_ базового и производного класса. Обычно для этого вызывают \_\_init\_\_ для *базового класса* в самом начале инициализирующего метода *производного класса*:

**class** С1:

 **def** \_\_init\_\_(*self*, x): *self*.x = x

**class** С2(С1):

 **def** \_\_init\_\_(*self*, x, y): *self*.\_\_init\_\_(x); *self*.y = y

Но это приведет к рекурсии: метод С2.\_\_init\_\_ создаётся на этапе объявления класса, а значит при поиске *self*.\_\_init\_\_ найдется именно С2.\_\_init\_\_ (у экземпляра *self* такого атрибута нет, а значит поиск идёт в его классе), а не С1.\_\_init\_\_.

Выход из этой ситуации — вызвать метод С1.\_\_init\_\_ явно.

Можно использовать встроенную функцию **super**. Как пример ‑ следующий код:

**class** C1:

 **def** \_\_init\_\_(*self*, x): **print**("Инициализация 1"); *self*.x = x

**class** C2(C1):

 **def** \_\_init\_\_(*self*, x, y):

 **print**("Инициализация 2"); **super**().\_\_init\_\_(x); *self*.y = y

b = C2(42, 3.14); **print**(b.x, b.y)

**Полиморфизм** – концепция ООП, когда атрибут (метод или свойство) используют для представления различных типов в различных сценариях использования, обеспечивая возможность понимания связанных объектов как подобных. Это позволяет базовому классу определять общие компоненты для всех наследников.

Дочерний класс при необходимости может переназначить методы базового. Здесь по существу переопределяется реакция на один и тот же запрос. Например, в Python есть некоторые функции, которые могут принимать аргументы разных типов. Одна из таких функций – **len**(), то есть имеет место полиморфизм.

Рассмотрим пример реализации полиморфизма в Python.

**from** math **import** pi

**class** Shape:

 **def** \_\_init\_\_(*self*, name): *self*.name = name

 **def** area(*self*): **pass**

 **def** fact(*self*): **return** "Это фигура на плоскости"

 **def** \_\_str\_\_(*self*): **return** *self*.name

**class** Square(Shape):

 **def** \_\_init\_\_(*self*, length): **super**().\_\_init\_\_("Square"); *self*.length = length

 **def** area(*self*): **return** *self*.length\*\*2

 **def** fact(*self*): **return** "В квадрате каждый угол равен 90 градусов"

**class** Circle(Shape):

 **def** \_\_init\_\_(*self*, radius):

 **super**().\_\_init\_\_("Circle")

 *self*.radius = radius

 **def** area(*self*): **return** pi\**self*.radius\*\*2

a = Square(4); b = Circle(7); **print**(b); **print**(b.fact()); **print**(a.fact()); **print**(b.area())

Чтобы исключить возможность обращения напрямую к атрибутам, их делают «закрытыми». В Python возможны следующие варианты доступа к данным:

* attribute (без подчеркиваний в начале) – **public**;
* \_attribute (с подчеркиванием) – **protected** (внутри класса и его дочерних);
* \_\_attribute (2 подчеркивания) – **private** (внутри класса, без наследования).

***Практическое занятие 3(1 час)***

1.Напишите программу, которая описывает класс СPoint – точка. На его основе создайте класс CLine. Все классы должны иметь методы для установки и получения значений всех координат, а также изменения названия.

**class** CPoint:

 **def** \_\_init\_\_(*self*, nam,x,y): *self*.nam=nam; *self*.x = x; *self*.y=y

 **def** printP(*self*): **print**("Точка ", *self*.nam, "(", *self*.x,",",*self*.y,")" )

**class** CLine(CPoint):

 **def** \_\_init\_\_(*self*,nam,x,y,x1,y1): **super**().\_\_init\_\_(nam,x,y); *self*.x1=x1; *self*.y1=y1

 **def** printL(*self*):**print**("Отрезок",*self*.nam,"(",*self*.x,",",*self*.y,",",*self*.x1,",", *self*.y1,")")

p1=CPoint("A",6,7); p1.printP(); L1=CLine("D",6,8,0,9); L1.printL()

2.Напишите программу, которая описывает класс Animal – животное, с полями вес и возраст. На его основе создайте класс Dog – собака с полями кличка и порода. Все классы должны иметь методы для установки и получения значений всех полей.

**class** Animal:

 **def** \_\_init\_\_(*self*,age, weight): *self*.age=age; *self*.weight = weight;

 **def** printA(*self*): **print**("Животное ", *self*.age, *self*.weight)

**class** Dog(Animal):

 **def** \_\_init\_\_(*self*, age, weight, nam, kind):

 **super**().\_\_init\_\_(age, weight); *self*.nam=nam; *self*.kind=kind

 **def** printD(*self*): **print**("Собака ", self.nam, "весит ",*self*.weight)

d1=Dog(2,10,"Pet","мопс"); d1.printD()

3.Напишите программу, которая описывает класс Point – точка. На его основе создайте класс Сirc. Добавьте методы вычисления длины окружности.

**import** math

**class** Point:

 **def** \_\_init\_\_(*self*, nam,x,y): *self*.nam=nam; *self*.x = x; *self*.y=y

 **def** printP(*self*): **print**("Точка ", *self*.nam, "(", *self*.x,",",*self*.y,")" )

**class** Circ(Point):

 **def** \_\_init\_\_(*self*,nam,x,y,r): **super**().\_\_init\_\_(nam,x,y); *self*.r=r

 **def** printC(*self*): **print**("окружность ", *self*.nam, "(", *self*.x,",",*self*.y,")", *self*.r)

 **def** lenC(*self*): **return** 2\*math.**pi**\**self*.r

c1=Circ("O",3,3,15); c1.printC(); **print**(c1.lenC())

4.Напишите программу для класса Man, с двумя полями – имя и год рождения. На его основе создайте класс Student – студент с полями курс и специальность. Все классы должны иметь методы для установки и получения значений всех полей.

**class** Man:

 **def** \_\_init\_\_(*self*, FIO, year): *self*.FIO=FIO; *self*.year=year

 **def** printM(*self*): **print**("Человек ", *self*.FIO, *self*.year,"Года рождения")

**class** Student(Man):

 **def** \_\_init\_\_(*self*, FIO, year,kurs,spec):

 **super**().\_\_init\_\_(FIO, year); *self*.kurs=kurs; *self*.spec=spec

 **def** printS(*self*): **print**("Студент ", *self*.FIO, " на ", *self*.kurs,"курсе");

m1=Man("Петров",1980);m1.printM();

s1=Student("Иванов",1998,1,"090302"); s1.printS()

***Лекционное занятие 4(1 час)***

Рассмотрим несколько примеров описания классов в языке Python.

Пример 1. Разработка класса для решения уравнения: $ax^{2}+bx+c=0, a\ne 0.$

Здесь поля - коэффициенты a,b,c, а методы \_\_init\_\_ и метод вычисления корней.

**class** SquareEquation:

 **def** \_\_init\_\_(*self*, a, b, c): *self*.a = a; *self*.b = b; *self*.c = c

 **def** Calc(*self*):

 d = *self*.b\*\*2 - 4\**self*.a\**self*.c

 **if** (d<0): **print**("The equation has no roots"); **return** []

 **else**: x1=(-*self*.b-d\*\*0.5)/(2.0\**self*.a);x2=(-*self*.b+d\*\*0.5)/(2.0\**self*.a);**return** [x1, x2]

# x1, x2 - локальные переменные - не члены данных класса

q=SquareEquation(1,1,-2);result=q.Calc();**print**('корни:','x1=',result[0],'x2=',result[1])

Обратите внимание, что Calc возвращает результат в виде списка.

|  |  |
| --- | --- |
| Рис. 8. Результат работы программы: |  |

Пример 2. Разработка родительского класса Фигура и наследника Прямоугольник.

Фигура будет содержать свойства цвет, конструктор, к наследнику добавляются ширина, высота и метод, выводящий информацию о прямоугольнике.

|  |  |
| --- | --- |
| 1234567891011121314151617181920212223242526 | **class** Figure: **def** \_\_init\_\_(*self*, color): *self*.\_\_color = color @**property** **def** color(*self*): **return** *self*.\_\_color @color.**setter** **def** color(*self*, c): *self*.\_\_color = c **def** info(*self*): **print**("Figure"); **print**("Color: " + *self*.\_\_color)**class** Rectangle(Figure): **def** \_\_init\_\_(*self*,w,h,c):**super**().\_\_init\_\_(c);*self*.\_\_w=w; *self*.\_\_h=h @**property** **def** width(*self*): **return** *self*.\_\_w @width.**setter** **def** width(*self*, w): **if** w > 0: *self*.\_\_w = w **else**: **raise** **ValueError** @**property** **def** height(*self*): **return** *self*.\_\_h @height.**setter** **def** height(*self*, h): **if** h > 0: *self*.\_\_h = h **else**: **raise** **ValueError** **def** area(*self*): **return** *self*.\_\_w \* *self*.\_\_h **def** info(*self*): **print**("Rectangle","Color:"+***self*.color**,"Width:" + **str**(*self*.\_\_w), sep='\n') **print**("Height:" + **str**(*self*.\_\_h),"Area:" + **str**(*self*.area()), sep='\n')rect = Rectangle(10, 12, "green"); rect.height=20; rect.info() |
| Для удобства ссылки все строки пронумерованы Обратите внимание на ***self*.color…** Рис. 9. Результат работы программы: |  |

Обратите внимание и на метод \_\_init\_\_ класса Прямоугольник, где вызван конструктор (это не совсем верно, но будем говорить так) его родительского класса.

Для объявления метода свойством используется *декоратор* @property. Здесь преимуществом работы через свойства является проверка данных до присвоения атрибутам. В примере, если значение <0, то сработает исключение **ValueError**.

Пример 3. Разработка класса Рациональная дробь.

Атрибуты: числитель, знаменатель, \_\_init\_\_, методы умножения и деления, а также статический генератор случайной дроби в заданном диапазоне целых чисел.

|  |  |
| --- | --- |
| 1234567891011121314151617181920 | **from** math **import** **gcd****from** random **import** **randint****class** My\_Frac: **def** \_\_init\_\_(*self*, n, d): **if** n != 0 **and** d != 0: k = **gcd**(n, d); *self*.n=n//k; *self*.d=d//k **else**: **raise** **ValueError** @**staticmethod** **def** generate(n\_min, n\_max, d\_min, d\_max): **return** My\_Frac(**randint**(n\_min, n\_max), **randint**(d\_min, d\_max)) **def** \_\_str\_\_(*self*): **return** f'{*self*.n}/{*self*.d}' **def** \_\_mul\_\_(*self*, other):  **if** **isinstance**(other,My\_Frac): **return** My\_Frac(*self*.n\*other.n,*self*.d\*other.d) **if** **isinstance**(other,int): **return** My\_Frac(*self*.n\*other, *self*.d) **return** *self*  **def** \_\_truediv\_\_(*self*, other):  **if** **isinstance**(other,My\_Frac): **return** My\_Frac(*self*.n\*other.d, *self*.d\*other.n) **if** **isinstance**(other,int): **return** My\_Frac(*self*.n, *self*.d\*other) **raise** **TypeError** d1=My\_Frac.generate(1,9,1,9); d2=My\_Frac.generate(1,9,1,9)cm=d1\*d2;**print**(f'{d1}\*{d2}={cm}') |
| Рис. 10. Результат работы программы: |  |

Обратите внимание, что конструктор содержит сокращение дроби!

 Остановимся подробнее на работе декораторов, являющихся удобным инструментом современного программирования во всех развивающихся языках. Это один из инструментов *функционального* программирования, включенный в современные *процедурные* языки, поскольку это не влияет на быстродействие программ, а точнее – снижение быстродействия в этом случае не столь существенно, как выигрыш от полученных инструментов для программирования.

 Декоратор – функциональная надстройка, принимающая функции в качестве аргументов, а затем, не изменяя их тела, добавляет к ним новые возможности. В Python существует некоторый набор предопределенных декораторов, в частности, остановимся подробнее на используемых здесь **@property** и **@staticmethod**.

 **@property** определяет совокупность свойств объекта, где уже определены внутренние декораторы setter, getter и deleter. Рассмотрим это в примере 2.

В строке 2 конструктору передано некоторое значение с именем color, которое присваивается **private** – переменной \_\_color класса Figure.

В строках 3-4 реализовано **public** - *свойство* color, которое равно \_\_color, то есть мы имеем два одинаковых свойства с разным доступом! Здесь декоратор @property

работает одновременно как getter, то есть возвращает значение, а setter определен в строках 5-6, причем строка 5 - setter-декоратор для только что созданного свойства color, строка 6 процедурой color(c) присваивает \_\_color значение своего аргумента. Строка 7 содержит процедуру info вывода некоторой информации класса Figure.

 Класс Rectangle наследуется от Figure (8), но здесь в конструкторе уже три аргумента – w, h, c, которые присваиваются переменным \_\_w, \_\_h этого класса, а также, через super() – обращение, переменной \_\_color класса Figure (9). Далее в строках 10-21 определены свойства width и height с контроем ошибки. Отметим, что эти свойства public-уровня класса Rectangle, наряду с которыми существуют private-свойства \_\_w и \_\_h, к которым обращается (22) функция area(). В строках 23-25 определена public-процедура info() вывода информации класса Rectangle, перезагружающая info-процедуру родительского класса (*полиморфизм*).

 Декоратор **@staticmethod** определяет набор элементов интерфейса класса, являющихся статическими, что означает постоянство ссылок на данный компонент интерфейса при обращении, а потому к ним можно обращаться через класс без создания объекта. Рассмотрим работу этого декоратора в Примере 3.

 После импорта методов gcd и randint библиотек math и random (строки 1-2) выполняется строка 19, где создаются объекты d1, d2 класса My\_Frac, которые возвращаются методом generate класса My\_Frac без предварительного создания объекта. Это возможно, поскольку декоратор объявил метод статическим (7) и обращение к нему аналогично обращению к одноименной процедуре, но всё же как к методу класса My\_Frac, поскольку метод описан внутри него (8). Метод задает два случайных числа заданных отрезках, возвращая объект My\_Frac с ними в качестве аргументов, что запускает конструктор класса (4), где эти числа сокращаются на их НОД и присваиваются соответствующим переменным класса, возвращая объект. В классе *переопределены стандартные методы преобразования объекта* класса в строку \_\_str\_\_ (10), деления \_\_truediv\_\_ (15) и произведения \_\_mul\_\_ (11) двух объектов: строка 12 – умножение My\_Frac на My\_Frac, 13 - умножение int на int, 16 – деление My\_Frac на My\_Frac, 17 - деление int на int. В каждой из них функция isinstance проверяет соответствие типов, возвращая объект My\_Frac, генерируя ошибку, если типы не совпадают при делении (18) и при умножениии не-My\_Frac на My\_Frac (автоматически). При умножении My\_Frac на int функция работает корректно. Это используется в строке 20, где сначала идет обращение к методу \_\_mul\_\_, а в команде print – к методу \_\_str\_\_.

 Это было краткое знакомство с декораторами, а подробнее – самостоятельно! *В последующих примерах практических заданий декораторы не использованы.*

***Практическое занятие 4(1 час)***

1.Напишите класс для преобразования целого числа в римскую цифру.

**class** Rom:

 **def** int\_to\_Roman(*self*, num):

 syb=["M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV","I"]

 val=[1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1]; roman\_num = ''; i = 0

 **while** num > 0:

 **for** \_ **in** **range**(num//val[i]): roman\_num += syb[i]; num -= val[i]

 i += 1

 **return** roman\_num

**print**(Rom().int\_to\_Roman(1)); **print**(Rom().int\_to\_Roman(4000))

2.Напишите класс поиска пар элементов массива X, с суммой равной T.

**class** PySol:

 **def** twoSum(*self*, X, T):

 lookup = {}

 **for** i, num **in** **enumerate**(X):

 **if** T - num **in** lookup: **return** (lookup[T - num], i ) #первый подходящий

 lookup[num] = i

**print**("index1=%d, index2=%d" % PySol().twoSum((10,20,10,40,50,60,70),50))

Интереснее программа поиска номеров всех пар с искомой суммой:

**class** PySol:

 **def** \_\_init\_\_(*self*, T): *self*.T=T

 **def** twoSum(*self*, X):

 rez = []

 **for** i **in** range(0,len(X)-1):

 **for** j **in** range(i+1,len(X)):

 **if** X[i]+X[j]==*self*.T: rez.append((i,j))

 **return** rez

p=PySol(30);**print**(f"Пары:{p.twoSum((10,20,10,40,50,20,70))}-дают в сумме {p.T}")

3.Напишите класс Car, где определены методы запуска двигателя и общего пробега.

**class** Car: #здесь пример с использованием кириллицы

 **def** \_\_init\_\_(*self*, цвет, расход, бак, пробег=0):

 *self*.цвет = цвет; *self*.расход = расход; *self*.бак = бак

 *self*.запас = бак; *self*.пробег = пробег; *self*.заведен = **False**

 **def** завести(*self*):

 **if** **not** *self*.заведен **and** *self*.запас>0: *self*.заведен=**True**; **return** "Движок заведен"

 **return** "Движок уже был заведен."

 **def** заглушить(*self*):

 **if** *self*.заведен: *self*.заведен = **False; return** "Движок заглушен."

 **return** "Движок уже был заглушен."

 **def** drive(*self*, distance):

 **if** **not** *self*.заведен: **return** "Движок не заведен."

 **if** *self*.запас / *self*.расход \* 100 < distance: **return** "Мало топлива."

 *self*.пробег += distance; *self*.запас -= distance / 100 \* *self*.расход

 **return** f"Проехали {distance} км. Остаток топлива: {self.запас} л."

 **def** заправить(*self*): *self*.запас = *self*.бак

 **def** get\_пробег(*self*): **return** *self*.пробег

 **def** get\_запас(*self*): **return** self.запас

 **def** get\_расход(*self*): **return** self.расход

car\_1=Car(цвет="black", расход=10, бак=55); car\_1.завести(); **print**(car\_1.drive(30))

4.Создать класс поиска площади и периметра прямоугольника по его сторонам.

**class** Rectangle():

 **def** \_\_init\_\_(*self*, L, W): *self*.length = L; *self*.width = W

 **def** printRec(*self*): **print**("Размеры прямоугольника - ", *self*.length,'\*', *self*.width)

 **def** rectangle\_area(*self*): **return** *self*.length\**self*.width #возвращает площадь

 **def** rectangle\_per(*self*): **return** 2\*(*self*.length+*self*.width) #возвращает периметр

r = Rectangle(12, 10); r.printRec(); print('S=',r.rectangle\_area(),'P=',r.rectangle\_per())

***Практическое занятие 5(2 час)***

1.Создать класс «Комплексное число».

**class** ComplexNumber:

 **def** \_\_init\_\_(*self*, r=0, i=0): *self*.real = r; *self*.imag = i

 **def** get\_data(*self*): **print**(f'{*self*.real}+{*self*.imag}j')

 **def** get\_abs(*self*): **return** (*self*.real\*\*2+ *self*.imag\*\*2)\*\*0.5

num1 = ComplexNumber(2, 3); num1.get\_data(); print(num1.get\_abs())

2.Создать класс «Студент», содержащий имя, номер группы и список оценок.

**class** Student:

 **def** \_\_init\_\_(*self*, nm="", gr=0, ms=[]): *self*.nm=nm; *self*.gr=gr; *self*.ms=ms

 **def** **\_\_**str**\_\_**(*self*):

 txt = 'Студент: ' + *self*.nm + ' Группа: ' + **str**(*self*.gr) + ' Оценки:'

 **for** x **in** *self*.ms: txt += ' ' + **str**(x)

 **return** txt

st1=Student("Иванов",6,[5,4,3,5]); print(st1.\_\_str\_\_())

3.Создать класс Robot, со свойствами-координатами Робота в отрезке [0, 100], и методом move его смещения на 1 клетку по горизонтали или вертикали.

**class** Robot:

 **def** \_\_init\_\_(*self*,x,y): *self*.x=x; *self*.y=y

 **def** move(*self*,dx,dy):#возможно перемещение по x или y на 1 в пределах [0,100]

 **if** dx **in** **range**(-1,2) **and** dy **in** **range**(-1,2) **and** dx\*dy==0:

 **if** *self*.x+dx **in** **range**(0,101) **and** *self*.y+dy **in** **range**(0,101): *self*.x+=dx; *self*.y+=dy

 **def** printkor(*self*): **print**("координаты робота:", *self*.x, *self*.y)

r1=Robot(10,10); r1.printkor(); r1.move(-1,0); r1.printkor()

4.Создать класс Оборудование, наследовав его в Принтер, Сканер и Ксерокс. В Принтер перегрузить \_\_str()\_\_, а класс Действие() перегружать во всех дочерних.

**class** Оборудование:

 **def** \_\_init\_\_(*self*, nm, mk, y): *self*.nm=nm; *self*.mk=mk; *self*.y=y

 **def** action(*self*): **return** 'Действие не определено'

 **def** \_\_str\_\_(*self*): **return** f'{*self*.nm} {*self*.mk} {*self*.y}'

**class** Принтер(Оборудование):

 **def** \_\_init\_\_(*self*, ser, nm, mk, y): **super**().\_\_init\_\_(nm, mk, y); *self*.ser = ser

 **def** \_\_str\_\_(*self*): **return** f'{*self*.nm} {*self*.ser} {*self*.mk} {*self*.y}'

 **def** action(*self*): **return** 'Печатает'

**class** Сканер(Оборудование):

 **def** \_\_init\_\_(*self*, nm, mk, y): **super**().\_\_init\_\_(nm, mk, y)

 **def** action(*self*): **return** 'Сканирует'

**class** Ксерокс(Оборудование):

 **def** \_\_init\_\_(*self*, nm, mk, y): **super**().\_\_init\_\_(nm, mk, y)

 **def** action(*self*): **return** 'Копирует'

sklad = []; sc = Сканер('Mustek','BP 1200CU', 2010); sklad.append(sc)

xer = Ксерокс('Xerox','Phaser', 2019); sklad.append(xer)

prn=Принтер("1200",'hp', 'LJ', 2018); sklad.append(prn);**print**("Есть на складе:")

**for** x **in** sklad: **print**(x, x.action())

**for** x **in** sklad:

 **if** **isinstance**(x,Принтер): sklad.remove(x)

**print**("\nНа складе осталось:")

**for** x **in** sklad: **print**(x, x.action())