



Лекции. Практические занятия

Солдатов Е.Ю.

2025 г.

МНОЖЕСТВА

Ещё одним типом элементов являются множества (**set**). Для определения используются фигурные скобки.

```
cards = {"Король", "Дама", "Туз", "Валет", "Туз"}  
print(cards)
```

```
D:\python>python code.py  
{'Дама', 'Туз', 'Король', 'Валет'}
```

Множество содержит только уникальные значения!

Задать множество можно функцией `set()`, передав в неё список или кортеж:

```
cards = set(["Король", "Дама", "Туз", "Валет", "Туз"])
```

Для определения длины множества – функция `len()`.

Для добавления элементов – метод `add()`.

Для удаления – метод `remove()` или `discard()`. Второй не выдаст ошибку при отсутствии элемента в множестве.

Удалить все элементы можно с помощью метода `clear()`. Получится пустое множество.

Множества можно копировать методом `copy()`, объединять методом `union()`:

```
cards1 = {"Король", "Дама", "Туз", "Валет", "Туз"}  
cards2 = {"Король", "Десятка", "Туз", "Валет", "Джокер"}  
cards3 = cards1.union(cards2)  
print(cards3)
```

```
D:\python>python code.py  
{'Валет', 'Джокер', 'Дама', 'Король', 'Туз', 'Десятка'}
```

МНОЖЕСТВА

Пересечение 2 множеств можно получить методом *intersection()*:

```
cards1 = {"Король", "Дама", "Туз", "Валет", "Туз"}  
cards2 = {"Король", "Десятка", "Туз", "Валет", "Джокер"}  
cards3 = cards1.intersection(cards2)  
print(cards3)
```

```
D:\python>python code.py  
{ 'Король', 'Туз', 'Валет' }
```

print(cards1 & cards2) # Выдаст то же самое.

Метод *difference()* возвратит разность множеств:

```
cards3 = cards1.difference(cards2)
```



```
D:\python>python code.py  
{ 'Дама' }
```

Выяснить, является ли множество подмножеством или супермножеством другого можно так:

```
print(cards3.issubset(cards1))  
print(cards3.issuperset(cards1))
```

```
D:\python>python code.py  
True  
False
```

Тип **frozen set** является видом множеств, которое не может быть изменено. От обычных множеств отличается так же как кортеж от списков: менее гибкие, зато более быстрые.

Для его создания используется функция *frozenset()*:

```
cards = frozenset(["Король", "Дама", "Туз", "Валет", "Туз"])
```

РАЗНОЕ: ФУНКЦИИ

Функция также может принимать переменное количество позиционных аргументов, тогда перед именем ставится *:

```
def func(*args):  
    return args
```



```
D:\python>python code.py  
(1, 2, 3, 'abc')
```

```
print(func(1, 2, 3, 'abc'))
```

Как видно из примера, args - это кортеж из всех переданных аргументов функции, и с переменной можно работать также, как и с кортежем.

Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится **:

```
def func(**kwargs):  
    return kwargs
```



```
D:\python>python code.py  
{ 'a': 1, 'b': 2, 'c': 3 }
```

```
print(func(a=1, b=2, c=3))
```

В переменной kwargs у нас хранится словарь, с которым мы, опять-таки, можем делать все, что нам заблагорассудится.

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции **lambda**. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией def func():

```
func = lambda x, y: x + y
```

```
print(func(1, 2))
```

```
print((lambda x, y: x + y)(5, 6))
```



```
D:\python>python code.py  
3  
11
```

lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же.

РАЗНОЕ: ГЕНЕРАТОРЫ СПИСКОВ

List comprehensions

Допустим, нам необходимо получить список нечетных чисел, не превышающих 25.

Обычно мы сделали бы это так:

```
print(list(range(1, 25, 2)))
```

Но можно то же самое сделать так:

```
print([x for x in range(1, 25, 2)])
```

```
D:\python>python code.py
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
```

Можно добавить дополнительные условия фильтрации. Например, доработаем наш предыдущий пример так, чтобы исключались квадраты чисел, кратных 3:

```
print([x**2 for x in range(1, 25, 2) if x % 3 != 0])
```

```
D:\python>python code.py
[1, 25, 49, 121, 169, 289, 361, 529]
```

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Python поддерживает объектно-ориентированную парадигму программирования, а это значит, что мы можем определить компоненты (модули) программы в виде **классов**.

Класс – аналог типа переменной в ООП. Тут это шаблон **объекта**.

Класс объединяет набор функций и переменных, которые выполняют определенную задачу. Функции класса - **методы**. Переменные **класса** – **атрибуты**.

Создадим класс и попробуем с ним поработать:

```
class Particle:  
    name = "Частица"  
    charge = 0  
    def display_info(self):  
        print("Эта частица - ", self.name)
```

```
particle1 = Particle()  
particle1.name = "Электрон"  
particle1.display_info()
```



```
D:\python>python code.py  
Эта частица - Электрон
```

Мы создали класс с 2 атрибутами и одним методом. Создаём пустой объект, записываем значение в атрибут и исполняем метод.

Методы любого класса должны принимать в качестве первого параметра ссылку на текущий объект, **self** (в си++ аналог - this). Через **self** внутри класса можно обратиться к методам/атрибутам этого же класса.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Для создания объекта класса используется конструктор. На предыдущем слайде мы использовали конструктор по умолчанию, который неявно

- имеют все классы: `Particle()`

Определим теперь конструктор явно:

```
class Particle:
```

```
    # конструктор
```

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

```
        self.name = name # устанавливаем атрибут класса: имя частицы
```

```
        self.charge = charge # устанавливаем атрибут класса: заряд
```

```
    def display_info(self):
```

```
        print("Эта частица - ", self.name, "\nЕё заряд: ", self.charge)
```

```
particle1 = Particle("Электрон", -1)
```

```
particle1.display_info()
```

```
D:\python>python code.py
```

```
Эта частица - Электрон
```

```
Её заряд: -1
```

После окончания работы с объектом мы можем использовать оператор ***del*** для удаления его из памяти:

```
particle1 = Particle("Электрон", -1)
```

```
del particle1
```

Подключить класс из модуля `classes.py` можно обычно:

```
from classes import Particle
```

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Можно определить в классе деструктор, реализовав встроенную функцию `__del__`, который будет вызываться либо в результате вызова оператора `del`, либо при автоматическом удалении объекта (при окончании программы):

```
class Particle:
```

```
    # конструктор
```

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

```
        self.name = name # устанавливаем имя частицы
```

```
        self.charge = charge # устанавливаем заряд
```

```
    # деструктор
```

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

```
        print(self.name, "удален из памяти")
```

```
    def display_info(self):
```

```
        print("Эта частица - ", self.name, "\nЕё заряд: ", self.charge)
```

```
particle1 = Particle("Электрон", -1)
```

```
particle1.display_info()
```

```
del particle1 # удаление из памяти
```

```
D:\python>python code.py
Эта частица - Электрон
Её заряд: -1
Электрон удален из памяти
```

ООП: ИНКАПСУЛЯЦИЯ

По умолчанию атрибуты в классах являются общедоступными: из любого места программы можно получить атрибут объекта и изменить его.

• **Инкапсуляция** помогает разграничивать доступ к изменению атрибутов или к исполнению методов.

Для создания приватного атрибута в начале его наименования ставится двойной прочерк: `self.__name`. К такому атрибуту мы сможем обратиться только из того же класса.

class Particle:

конструктор

def __init__(self, name, charge):

self.__name = name # устанавливаем имя частицы

self.__charge = charge # устанавливаем заряд

def display_info(self):

print("Эта частица - ", self.__name, "\nЕё заряд: ", self.__charge)

particle1 = Particle("Электрон", -1)

particle1.display_info()

particle1.__name = "Протон" # Это ничего не сможет изменить

particle1.display_info()

```
D:\python>python code.py
Эта частица - Электрон
Её заряд: -1
Эта частица - Электрон
Её заряд: -1
```

ООП: ИНКАПСУЛЯЦИЯ

Однако все же нам может потребоваться устанавливать приватные атрибуты частицы извне. Для этого создаются свойства. Используя одно свойство, мы можем получить значение атрибута:

```
def get_charge(self):  
    return self.__charge
```

Такой метод часто называется геттер или аксессор.

Для изменения заряда определено другое свойство:

```
def set_charge(self, value):  
    if value in range(-2, 2):  
        self.__charge = value  
    else:  
        print("Недопустимый заряд!")
```

Данный метод еще называют сеттер или мьютейтор (mutator).

Необязательно создавать для каждого приватного атрибута подобную пару свойств. Так, в примере выше название частицы мы можем установить только из конструктора.

Эти 2 свойства можно ещё и аннотировать:

Для создания свойства-геттера над свойством ставится аннотация **@property**.

Для создания свойства-сеттера над свойством устанавливается аннотация **имя_свойства_геттера.setter**.

Тогда используется одно и то же имя для сеттера и геттера.

```
print(particle1.charge)
```

```
particle1.charge = -2
```

ООП: ИНКАПСУЛЯЦИЯ, СВОЙСТВА

```
class Particle:
```

```
    # конструктор
```

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

```
        self.__name = name # устанавливаем имя частицы
```

```
        self.__charge = charge # устанавливаем заряд
```

```
    def display_info(self):
```

```
        print("Эта частица - ", self.__name, "\nЕё заряд: ", self.__charge)
```

```
    @property # Нужно, чтобы использовать имя в подклассе
```

```
    def charge(self):
```

```
        return self.__charge
```

```
    @charge.setter
```

```
    def charge(self, value):
```

```
        if value in range(-2, 2):
```

```
            self.__charge = value
```

```
        else:
```

```
            print("Недопустимый заряд!")
```

```
particle1 = Particle("Электрон", -1)
```

```
print(particle1.charge)
```

```
particle1.charge=1
```

```
print(particle1.charge)
```

Программа выдаст:

-1

1

ООП: НАСЛЕДОВАНИЕ

Наследование позволяет создавать новый класс на основе уже существующего класса.

- Ключевыми понятиями наследования являются **подкласс** (наследник) и **суперкласс** (наследуемый).

```
class Particle:
```

```
    # конструктор
```

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

```
        self.__name = name # устанавливаем имя частицы
```

```
        self.__charge = charge # устанавливаем заряд
```

```
    def display_info(self):
```

```
        print("Эта частица - ", self.__name, "\nЕё заряд: ",
```

```
self.__charge)
```

```
    @property # Нужно, чтобы использовать имя в подклассе
```

```
    def name(self):
```

```
        return self.__name
```

```
class Boson(Particle): # Так реализовано наследование в Python
```

```
    def details(self, interaction):
```

```
        print(self.name, "переносит", interaction, "взаимодействие.")
```

```
particle2 = Boson("Глюон", 0)
```

```
particle2.details("сильное")
```

ООП: ПОЛИМОРФИЗМ

Полиморфизм даёт способность к изменению функционала, унаследованного от базового класса.

class Particle:

конструктор

def __init__(self, name, charge):

self.__name = name # устанавливаем имя частицы

self.__charge = charge # устанавливаем заряд

def display_info(self):

print("Эта частица - ", self.__name, "\nЕё заряд: ",

self.__charge)

@property

def name(self):

return self.__name

class Boson(Particle):

определение конструктора

def __init__(self, name, charge, interaction):

Particle.__init__(self, name, charge)

self.interaction = interaction

переопределение метода display_info

def display_info(self):

Particle.display_info(self)

print("Это бозон, переносящий", self.interaction, "взаимодействие.")

ООП: ПОЛИМОРФИЗМ

```
class Fermion(Particle):
    # определение конструктора
    def __init__(self, name, charge, is_stable):
        Particle.__init__(self, name, charge)
        self.is_stable = is_stable
    # переопределение метода display_info
    def display_info(self):
        Particle.display_info(self)
        print("Это фермион.")
        if self.is_stable:
            print("И он стабилен.")
        else:
            print("И он нестабилен.")

particle = [Particle("Электрон", -1), Boson("Глюон", 0, "сильное"),
            Fermion("Протон", +1, True)]

for part in particle:
    part.display_info()
    print()
```

ООП: ПОЛИМОРФИЗМ

```
D:\python>python code.py
Эта частица - Электрон
Её заряд: -1

Эта частица - Глюон
Её заряд: 0
Это бозон, переносящий сильное взаимодействие.

Эта частица - Протон
Её заряд: 1
Это фермион.
И он стабилен.
```

При работе с объектами бывает необходимо в зависимости от их типа (класса) выполнить те или иные операции. И с помощью встроенной функции ***isinstance()*** мы можем проверить тип объекта. Эта функция принимает два параметра: объект и тип (класс).

```
for part in particle:
    if isinstance(part, Boson):
        print(part.interraction)
    elif isinstance(part, Fermion):
        if part.is_stable:
            print("Стабилен")
    else:
        print(part.name)
print()
```



```
D:\python>python code.py
Электрон

сильное

Стабилен
```

ООП: ДЗ

Написать программу, которая распадает частицу с вероятностью, описанной в pdg (<http://pdg.lbl.gov/>). Взять как минимум 2-3 распада частицы. Общая вероятность всех распадов близка к 1.

Частица должна быть реализована в виде класса. Распад может быть реализован как метод класса или отдельный класс.

В итоге программа с правильной вероятностью выдаёт (печатает моду) тот или иной распад.

Должна быть возможность создать не 1 частицу, а массив частиц и получить число распадов в каждый канал.

Варианты по номеру в списке:

1) omega(782)

2) tau

3) top

4) pi+

5) pi-

6) pi0

7) eta0

8) Z

9) W+

10) W-

11) K+

12) Higgs

13) K_S

14) K_L

15) phi(1020)

16) f1(1285)

17) K1(1270)

18) D+

19) D0

20) B+

21) B0

22) f0(1500)

23) rho3(1690)

24) D*(2007)

25) D*(2010)

top->e nu_e b: 9

top->mu nu_mu b: 11

top->tau nu_tau b: 8

top->q q b: 72

Другой распад: 0

******Как дополнение можно программу сделать более универсальной, чтобы она работала с распадами не одной конкретной частицы, а с распадами нескольких частиц (можно вводить).