

Занятие 19

ООП

Что напечатают эти операторы?

```
def add_fruit(fruit, basket=[]):  
    basket.append(fruit)  
    return basket
```

```
b = add_fruit("banana")  
print(b)
```

```
c = add_fruit("apple")  
print(c)
```

Задача 18

Разработать систему решения задач учениками курса «Разработчик на Питоне».

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).

Разработайте систему атрибутов для каждого класса для хранения и использования описанных процессов.

Разработайте систему методов для каждого класса для решения описанных процессов.

Проверьте ее работу на одном учителе и на 2-3 учениках.

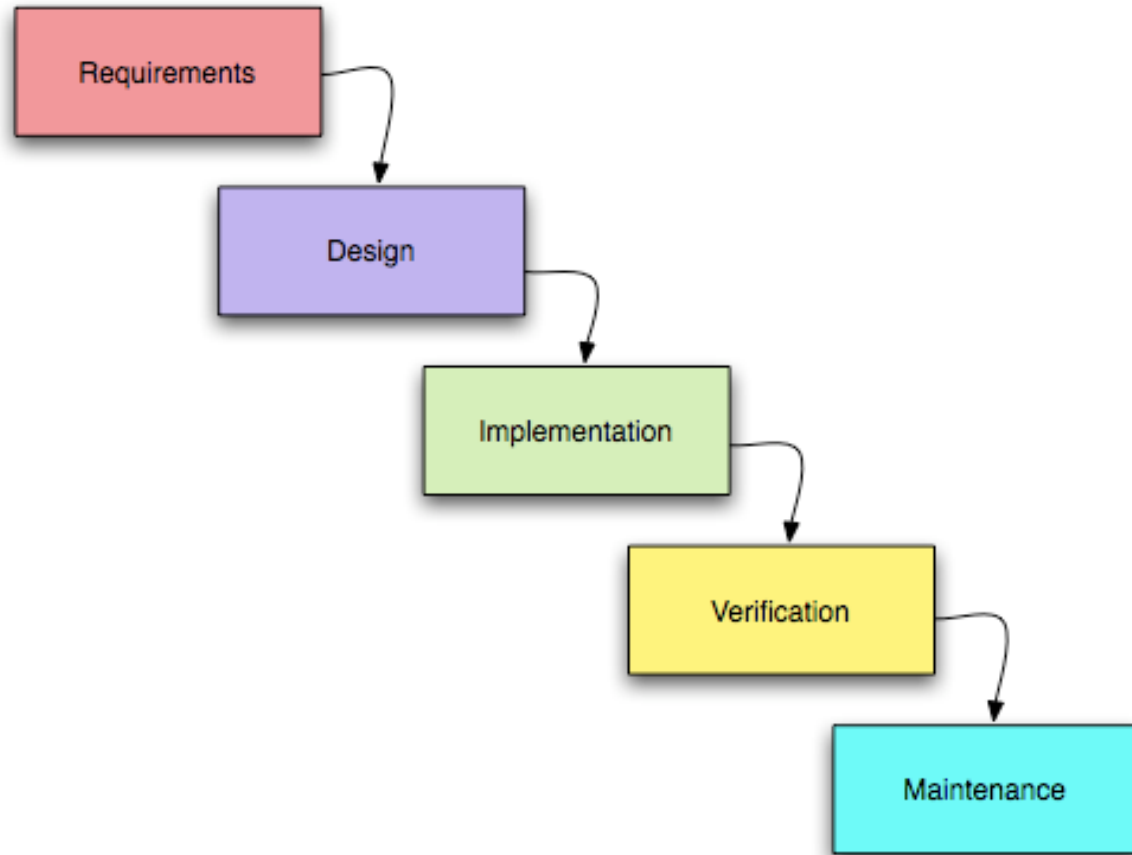
Проектирование

В ООП (и не только) очень важно предварительное проектирование.

Можно выделить следующие этапы разработки объектно-ориентированной программы:

1. Формулировка задачи (как можно более точная и детальная).
2. Определение объектов, участвующих в ее решении.
3. Проектирование классов, на основе которых будут создаваться объекты.
В случае необходимости установление между классами наследственных связей.
4. Определение ключевых для данной задачи свойств и методов объектов.
5. Создание классов, определение атрибутов: полей и методов.
6. Создание объектов.
7. Решение задачи путем организации взаимодействия объектов.

Каскадная модель – waterfall - водопад



- 1.Определение требований
- 2.Проектирование
- 3.Конструирование (также «реализация» либо «кодирование»)
- 4.Тестирование и отладка (также «**верификация**»)
- 5.Инсталляция, внедрение
- 6.Поддержка

Какие достоинства и какие недостатки?

Где до сих пор можно встретить такую модель?

Где до сих пор стоит придерживаться этой модели?

Как можно пытаться преодолеть недостатки?

Гибкая (agile) методология разработки - Scrum, Kanban и др.

Основополагающие принципы [Agile Manifesto](#)^{[5][6]}:

- наивысшим приоритетом признается **удовлетворение заказчика** за счёт ранней и бесперебойной поставки ценного программного обеспечения;
- **изменение требований** приветствуется даже в конце разработки (это может повысить конкурентоспособность полученного продукта);
- **частая поставка работающего программного обеспечения** (каждые пару недель или пару месяцев с предпочтением меньшего периода);
- общение представителей бизнеса с разработчиками должно быть **ежедневным** на протяжении всего проекта;
- проекты следует строить вокруг заинтересованных людей, которых следует обеспечить нужными условиями работы, поддержкой и доверием;
- самый эффективный метод обмена информацией в команде — **личная встреча**;
- работающее программное обеспечение — лучший измеритель прогресса;
- спонсоры, разработчики и пользователи должны иметь возможность поддерживать постоянный темп на **неопределённый срок**;
- постоянное внимание к техническому совершенству и хорошему проектированию увеличивают гибкость;
- простота, как искусство не делать лишней работы, очень важна;
- лучшие требования, архитектура и проектные решения получаются у самоорганизующихся команд;
- команда регулярно обдумывает **способы повышения своей эффективности** и соответственно корректирует рабочий процесс.

ООП в Python

Инкапсуляция, Наследование, Полиморфизм

- 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

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 ("Collected 20kg of {}".format(self.kind))`
- `f_tree = FruitTree("apple", 0.7)`
- `f_tree.give_fruits()`
- `f_tree.grow()`

Множественное наследование

- При множественном наследовании дочерний класс наследует все свойства родительских классов. Синтаксис множественного наследования очень похож на синтаксис обычного наследования.
- `class Horse: # А если определить метод Igogo?`
- `isHorse = True`
- `class Donkey: # А если определить метод Iaia?`
- `isDonkey = True`
- `class Mule(Horse, Donkey): # в каком порядке ищутся методы и атрибуты?`
- `pass`
- `mule = Mule()` # Какой из методов будет доступен? Igogo или Iaia?
- `mule.isHorse` # True
- `mule.isDonkey` # True

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: 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.

Если у объекта есть методы `__iter__` и `__next__`, то его можно итерировать

Если у объекта есть метод `__call__`, то можно его вызывать как функцию ()

Если у объекта есть метод `__getitem__`, то можно обращаться как к элементу коллекции []

Задание

Есть класс Student. Для каждого студента вводится его имя и количество пятерок, десятков и двадцаток, которые у него есть.

```
class Student:
    stdnt = []
    def __init__(self, name, fives, tens, twenties):
        self.name = name
        self.fives = fives
        self.tens = tens
        self.twenties = twenties
        Student.stdnt.append(self)
```

Напечатайте список студентов с указанием сколько у каждого денег.
Список можно накопить в свойстве класса stdnt

Два нижних подчеркивания (**dooble 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.
```

Магические методы в ООП

Table 1-1. Special method names (operators excluded)

Category	Method names
String/bytes representation	<code>__repr__</code> , <code>__str__</code> , <code>__format__</code> , <code>__bytes__</code>
Conversion to number	<code>__abs__</code> , <code>__bool__</code> , <code>__complex__</code> , <code>__int__</code> , <code>__float__</code> , <code>__hash__</code> , <code>__index__</code>
Emulating collections	<code>__len__</code> , <code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__contains__</code>
Iteration	<code>__iter__</code> , <code>__reversed__</code> , <code>__next__</code>
Emulating callables	<code>__call__</code>
Context management	<code>__enter__</code> , <code>__exit__</code>
Instance creation and destruction	<code>__new__</code> , <code>__init__</code> , <code>__del__</code>
Attribute management	<code>__getattr__</code> , <code>__getattribute__</code> , <code>__setattr__</code> , <code>__delattr__</code> , <code>__dir__</code>
Attribute descriptors	<code>__get__</code> , <code>__set__</code> , <code>__delete__</code>
Class services	<code>__prepare__</code> , <code>__instancecheck__</code> , <code>__subclasscheck__</code>

Методы конструирования и инициализации `__new__` `__init__`

Метод `__new__` вызывается первым.

Он открывает пространство памяти затем вызывается метод `__init__` .

```
class Person(object):  
    instance = None  
    def __new__(cls, *args, **kwargs):  
        if not cls.instance:  
            cls.instance = object.__new__(cls)  
        return cls.instance  
    def __init__(self):  
        self.__name = 'Peter I'
```

```
obj_one = Person()  
obj_two = Person()  
print( obj_one is obj_two )
```


Метод документирования `__doc__`

- `#` Дандер для документирования класса и методов
- `#` Комментарии нужно помещать сразу после объявления
- `class Test(object):`
- `""" Класс Test для демонстрации """`
- `def show(self):`
- `""" This is Show Function DocString """`
- `pass`
- `t = Test()`
- `print(t.__doc__)` `#` Описание класса
- `print(Test.__doc__)` `#` Описание класса
- `print(t.show.__doc__)` `#` Описание описание метода `#Давайте выполним это`

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}"

    def __repr__(self):

        return f"Car:{ self.model}, {self.color}, {self.VIN}"
```

```
car = Car("Mercedes-benz", "silver", "WDB1240221J081498")
```

```
print(car)
```

```
print(str(car))
```

```
print(repr(car))
```

```
# Что будет напечатано? А если не будет __str__? А если не будет __repr__?
```

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__`.

`__dict__` словарь для хранения атрибутов

```
class Car():
    color = "Red"
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __str__(self):
        return f"Модель: {self.model} с ценой {self.price} "

obj = Car("Mercedes-benz", 5_000_000)

print(obj.__dict__)          # {'model': 'Mercedes-benz', 'price': 5000000}
```

__dict__

Добавим атрибут

```
class Car:
```

```
    def __init__(self, model, price):  
        self.model = model  
        self.price = price
```

```
    def __str__(self):  
        return f"Модель: {self.model} с ценой {self.price} "
```

```
obj = Car("Mercedes-benz", 5_000_000)
```

```
obj.__dict__['color'] = "red"
```

```
print(obj.__dict__)
```

Задание

Переопределите метод `__str__`, чтобы в нем печатались все атрибуты объекта и их значения через запятую.

Например:

```
def __init__(self):  
    self.x = 0  
    self.y = 1
```

Должно быть напечатано `x:0,y:1`

Магические методы сравнения

`__eq__(self, other)` # Определяет поведение оператора равенства `==`

`__ne__(self, other)` # Определяет поведение оператора неравенства, `!=`

`__lt__(self, other)` # Определяет поведение оператора меньше, `<`

`__gt__(self, other)` # Определяет поведение оператора больше, `>`

`__le__(self, other)` # Определяет поведение оператора меньше или равно, `<=`

`__ge__(self, other)` # Определяет поведение оператора больше или равно, `>=`

Равенство значений объектов класса `__eq__`

```
class Car:
    def __init__(self, model, color, vin):
        self.model = model
        self.color = color
        self.VIN = vin

    def __eq__(self, obj):
        if not isinstance(obj, Car):
            raise ValueError("Передан другой тип объекта")
        return (self.VIN == obj.VIN)
```

```
car_one = Car("Mercedes-benz", "silver", "WDB1240221J081498")
car_two = Car("Mercedes-benz", "red", "WDB1240221J081498")
print(car_one == car_two)          # Что вернет сравнение ?
```

Что нужно для сравнения по всем параметрам ?

Задание

Задан класс, который принимает только один аргумент при создании объекта – целое число.

Переопределите функцию `eq`, чтобы равными считались два объекта, которые являются оба четными числами или оба нечетными числами.

Оператор больше `__gt__`

```
class Car:
```

```
    def __init__(self, model, price):  
        self.model = model  
        self.price = price
```

```
    def __gt__(self, obj):  
        if not isinstance(obj, Car):  
            raise ValueError("Передан другой тип объекта")  
        return (self.price > obj.price )
```

```
    def __str__(self):  
        return f"Модель: {self.model} с ценой {self.price} "
```

```
car_one = Car("Mercedes-benz", "5000000")
```

```
car_two = Car("Aurus Senat", "40000000")
```

```
print(car_one > car_two)
```

```
print(car_one < car_two)
```

```
# Что если изменить знак return (self.price < obj.price )
```

Задание

Класс объектов принимает одно значение в виде аргумента – строку.

Переопределите функцию `__gt__`, чтобы функция `min` выдавала объект с минимальной длиной строки.

Оператор меньше `__lt__`

```
class Car:
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def __lt__(self, obj):
        if not isinstance(obj, Car):
            raise ValueError("Передан другой тип объекта")
        return (self.price < obj.price )

    def __str__(self):
        return f"Модель: {self.model} с ценой {self.price} "

car_one = Car("Mercedes-benz", "50000000")
car_two = Car("Aurus Senat", "500000000")
print(car_one > car_two) ← ?
```

Задание

Есть класс Student. Для каждого студента вводится его имя и количество пятерок, десятков и двадцаток.

```
class Student:  
    def __init__(self, name, fives, tens, twenties):  
        self.name = name  
        self.fives = fives  
        self.tens = tens  
        self.twenties = twenties
```

```
# Определите метод __lt__ по количеству денег. Найдите студентов с  
минимальными и максимальными деньгами.
```

Математические операторы

`__add__()` — определяет поведение для сложения (оператор +)

`__sub__()` — определяет поведение для вычитания (оператор -)

`__mul__()` — определяет поведение для умножения (оператор *)

`__truediv__()` — определяет поведение для обычного деления (оператор /)

`__floordiv__()` — определяет поведение для целочисленного деления (оператор //)

`__mod__()` — определяет поведение для деления по модулю (оператор %)

Сложение

```
class Plus:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Plus(self.x + other.x, self.y + other.y)
    def __str__(self):
        return f"Plus({self.x}, {self.y})"

a = Plus(1,2)
b = Plus(10, 20)
print(a + b)  # Что напечатает?
```

Метод `__call__`

Данный метод позволяет экземплярам класса вести себя так, как будто они функции, то есть мы можем вызывать их, передавать их в функции, которые ожидают в качестве аргумента функцию, и так далее.

```
class LineGenerator:
    def __init__(self, k, b):
        self.k = k
        self.b = b

    def __call__(self, x):
        return self.k * x + self.b
```

```
line_func1 = LineGenerator(2, 5)      # получаем функцию  $y = 2x + 5$ 
line_func2 = LineGenerator(-6, 9)     # получаем функцию  $y = -6x + 9$ 
```

```
print(line_func1(10))                 # выводим значение  $2 \cdot 10 + 5 = 25$ 
print(line_func2(4))                  # выводим значение  $-6 \cdot 4 + 9 = -15$ 
```


Заключение

- Магические методы — это специальные методы, которые вызывается неявно.
- Также это подход python к перегрузке операторов, позволяющий классам определять свое поведение в отношении операторов языка.
- Из этого можно заключить, что интерпретатор имеет "некую таблицу" соответствия операторов к методам класса.
- Перегружая эти методы, вы можете управлять "поведением" операторов языка относительно вашего класса.

Итераторы

Итератор

Итератор (iterator) – это объект, который используется для прохода по итерируемому элементу.

В основном используется для коллекций (списки, словари, кортежи, строки и т.д)

Протокол iterator в Python включает две функции.

Один – `iter()`, другой – `next()`. И два дандера `__iter__` `__next__`

```
numbers = [1, 2, 3]
```

```
iterator = iter(numbers)          # создаем итератор на основе списка
```

```
print(next(iterator))             # запрашиваем и печатаем первый элемент итератора
```

```
print(next(iterator))             # запрашиваем и печатаем второй элемент итератора
```

```
print(next(iterator))             # запрашиваем и печатаем третий элемент итератора
```

```
print(next(iterator))             # возбуждается исключение StopIteration
```

Дандер __iter__

Рассмотрим пример:

```
num_list = [1, 2, 3, 4, 5]
```

```
for i in num_list:
```

```
    print(i)
```

```
dir(num_list)
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',  
 '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getitem__',  
 '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',  
 '__iter__', ...]
```

Дандер `__iter__`

Рассмотрим пример: `num_list = [1, 2, 3, 4, 5]`

1. Конструкция `for` в момент выполнения берет у коллекции объект `__iter__`

```
for i in num_list:
```

```
    print(i)
```

```
print(num_list.__iter__)
```

```
<method-wrapper '__iter__' of list object at 0x7f602f9b0bc0>
```

2. Получим объект итерации через функцию `iter()`

```
it = num_list.__iter__()
```

3. Посмотрим тип объекта

```
print(it) → <list_iterator object at 0x7f602f730d90>
```

```
dir(it)
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__init__', '__init_subclass__', '__iter__', '__new__',  
'__next__', ..]
```

Вывод.

Как мы могли убедиться, цикл **for** использует так называемые итераторы.

Коллекция содержит метод-wrapper **__iter__**, который в свою очередь возвращает объект итератор, в котором реализован метод **__next__**, который в свою очередь возвращает последующий итерируемый элемент.

Цикл **for** можно разложить на след. операции:

```
num_list = [1, 2, 3, 4, 5]
```

```
itr = iter(num_list)
```

```
print(next(itr))
```

```
print(next(itr))
```

```
print(next(itr))
```

```
print(next(itr))
```

```
print(next(itr))
```

```
print(next(itr))
```

Создание собственных итераторов

```
class SimpleIterator:
    def __init__(self, limit):
        self.limit = limit
        self.counter = 0

    def __next__(self):
        if self.counter < self.limit:
            self.counter += 1
            return self.counter
        else:
            raise StopIteration
```

```
s_iter1 = SimpleIterator(3)
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1)) → Что произойдет ?
```

Итератор для цикла

Для итерации в цикле нам необходимо реализовать дандер `__iter__` который возвращает `self`

```
class SimpleIterator:
```

```
    def __iter__(self):
```

```
        return self
```

```
    def __init__(self, limit):
```

```
        self.limit = limit
```

```
        self.counter = 0
```

```
    def __next__(self):
```

```
        if self.counter < self.limit:
```

```
            self.counter += 1
```

```
            return 1
```

```
        else:
```

```
            raise StopIteration
```

```
s_iter2 = SimpleIterator(5)
```

```
for i in s_iter2:
```

```
    print(i)
```


Итератор Factorials

Создадим бесконечный итератор Factorials, который генерирует последовательность факториалов всех натуральных чисел (от 1 до бесконечности).

Конструктор итератора не принимает аргументов.

```
class Factorials:
    def __init__(self):
        self.value = 1
        self.index = 1

    def __iter__(self):
        return self

    def __next__(self):
        self.value *= self.index
        self.index += 1
        return self.value
```

Задание

Создайте class, объект которого может генерировать последовательность чисел 1, 3, 6, 10, 15 и т.д., т.е. суммы чисел от 1 до n

```
import itertools
```

```
import itertools
```

```
dir(itertools)
```

```
# 'accumulate', 'chain', 'combinations', 'combinations_with_replacement',  
'compress', 'count', 'cycle', 'dropwhile', 'filterfalse', 'groupby', 'islice',  
'permutations', 'product', 'repeat', 'starmap', 'takewhile', 'tee', 'zip_longest'
```

itertools.permutations

```
import itertools
```

```
for x in itertools.permutations([1,2,3]):
```

```
    print(x, end = ' ')
```

```
# (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)
```

```
# А что будет, если в списке будет [1, 1, 2]
```

itertools.combinations

```
import itertools
for x in itertools.combinations([1,2,3,4], 3):
    print(x, end = ' ')
# (1, 2, 3) (1, 2, 4) (1, 3, 4) (2, 3, 4)
# А если второй параметр 1, 2 ,4?
```

itertools.combinations_with_replacement

```
import itertools
```

```
for x in itertools.combinations_with_replacement([1,2,3,4], 3):
```

```
    print(x, end = ' ')
```

```
 #(1, 1, 1) (1, 1, 2) (1, 1, 3) (1, 1, 4) (1, 2, 2) (1, 2, 3) (1, 2, 4) (1, 3, 3) (1,
3, 4) (1, 4, 4) (2, 2, 2) (2, 2, 3) (2, 2, 4) (2, 3, 3) (2, 3, 4) (2, 4, 4) (3, 3, 3)
(3, 3, 4) (3, 4, 4) (4, 4, 4)
```

itertools.cycle

```
import itertools
x = itertools.cycle([1,2,3,4])
for _ in range(10):
    print(next(x), end = ' ')
# 1 2 3 4 1 2 3 4 1 2
```

itertools.chain

```
import itertools
for x in itertools.chain([1,2,3], 'abc', {10:100, 20:200, 30:300}):
    print(x, end = ' ')
# 1 2 3 a b c 10 20 30
```


Задание

Дано слово. Составьте все всевозможные анаграммы этого слова без повторений.

Например, дано слово 'aabb'.

Результатом должна быть следующая последовательность:

{'baab', 'baba', 'abab', 'abba', 'aabb', 'bbaa'}

Задание

Напечатайте список студентов, отсортированных по сумме денег, имя и сумма денег.

Задача 19-1

В кошельке лежат бумажные купюры 50, 100, 200, 500, 1000, 5000 рублей, каждой купюры по одной штуке.

Какие суммы можно составить, если использовать этот набор купюр?

Подсказка. Используйте одну из наиболее подходящих функций модуля `itertools`.

Задача 19-2

Реализуйте класс `Fibonacci`, который реализует итератор, генерирующий бесконечную последовательность чисел Фибоначчи.

Например:

```
fibonacci = Fibonacci()  
print(next(fibonacci))  
print(next(fibonacci))  
print(next(fibonacci))  
print(next(fibonacci))
```

Должен печатать следующие числа:

```
1  
1  
2  
3
```

Задача 19-3

Определите класс Person. При создании объекта `p = Person('Иванов', 'Михаил', 'Федорович')` необходимо ввести полное имя человека.

При печати объекта должно выводиться следующее:

```
print(p) # чивородеФлиахиМвонави
```