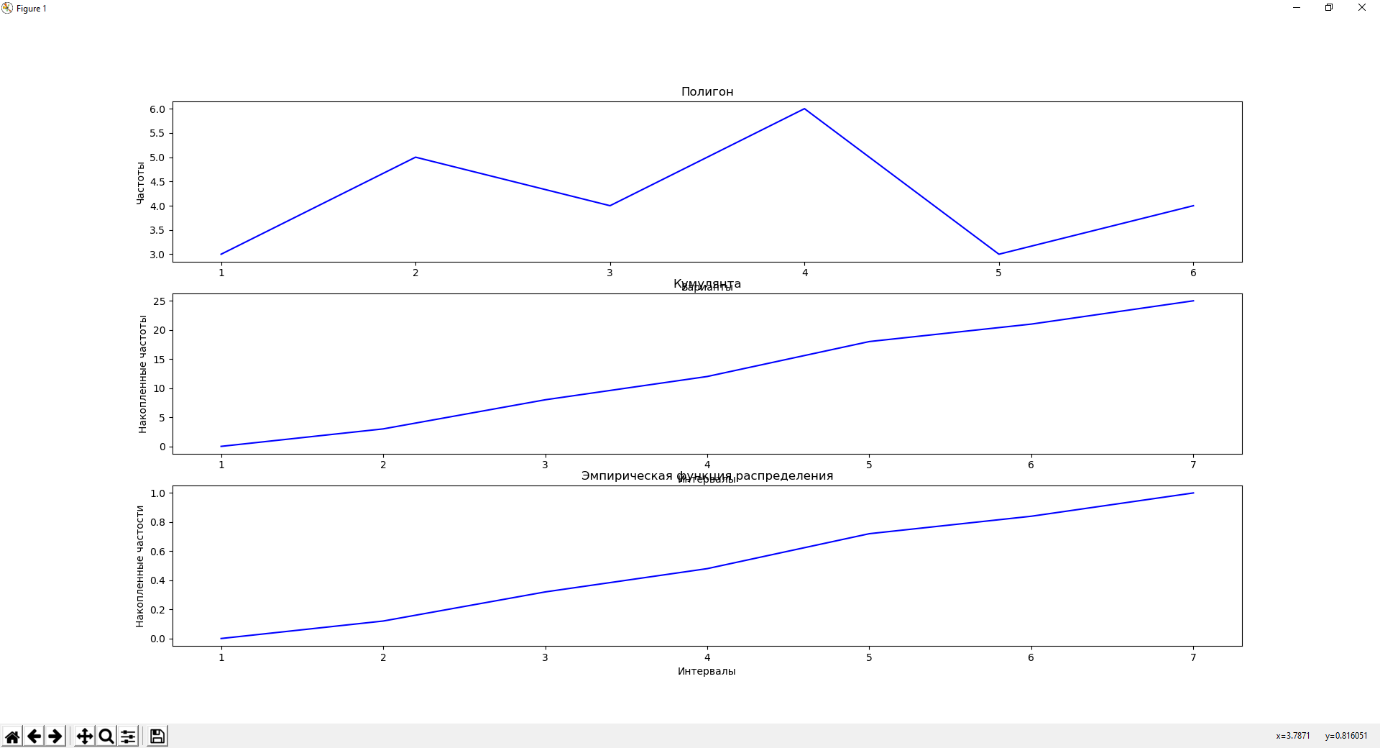
**Задачи к лабораторной работе №2**

# Задача №1

Постановка задачи: приводятся данные о распределении 25 работников одного из предприятий по тарифным разрядам: 4; 2; 4; 6; 5; 6; 4; 1; 3; 1; 2; 5; 2; 6; 3; 1; 2; 3; 4; 5; 4; 6; 2; 3; 4. Построить дискретный вариационный ряд и изобразить его графически.

Результаты:



Код программы:

from varseries import ContinuousVS, DiscreteVS  
  
  
def task1():  
 values = [  
 4, 2, 4, 6, 5, 6, 4, 1, 3, 1, 2, 5, 2, 6, 3, 1, 2, 3, 4, 5, 4, 6, 2, 3, 4  
 ]  
 v = DiscreteVS(values)  
 v.draw\_polygon()\  
 .draw\_cumulate()\  
 .draw\_empiric\_dist\_func()

# Задача №2

Постановка задачи: приведены данные о размерах вкладов 20 физических лиц в одном банке (тыс.руб): 60; 25; 12; 10; 68; 35; 2; 17; 51; 9; 3; 130; 24; 85; 100; 152; 6; 18; 7; 42. Построить интервальный вариационный ряд с равными интервалами.

Результаты:



Код программы:

def task2():  
 values = [  
 60, 25, 12, 10, 68, 35, 2, 17, 51, 9, 3, 130, 24, 85, 100, 152, 6, 18, 7, 42  
 ]  
 print(ContinuousVS(values).vs)

# Задача №3

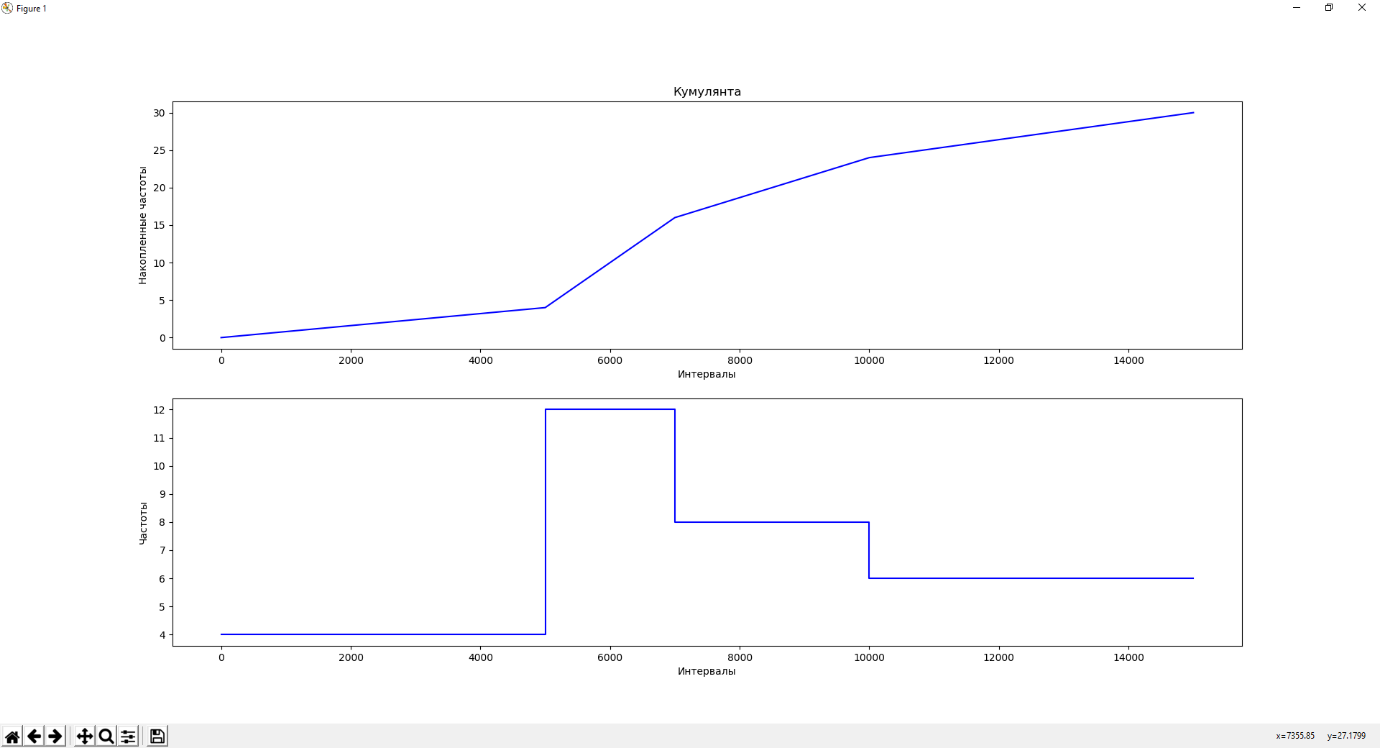
Постановка задачи: приводится распределение 30 работников фирмы по размеру месячной заработной платы



Построить и изобразить интервальный вариационный ряд графически в виде гистограммы и

кумуляты.

Результаты:



Код программы:

def task3():  
 values = {  
 (0, 5\*10\*\*3): 4, (5\*10\*\*3, 7\*10\*\*3): 12,  
 (7\*10\*\*3, 10\*\*4): 8, (10\*\*4, 1.5\*10\*\*4): 6  
 }  
 v = ContinuousVS(values)  
 v.draw\_cumulate()\  
 .draw\_hist()

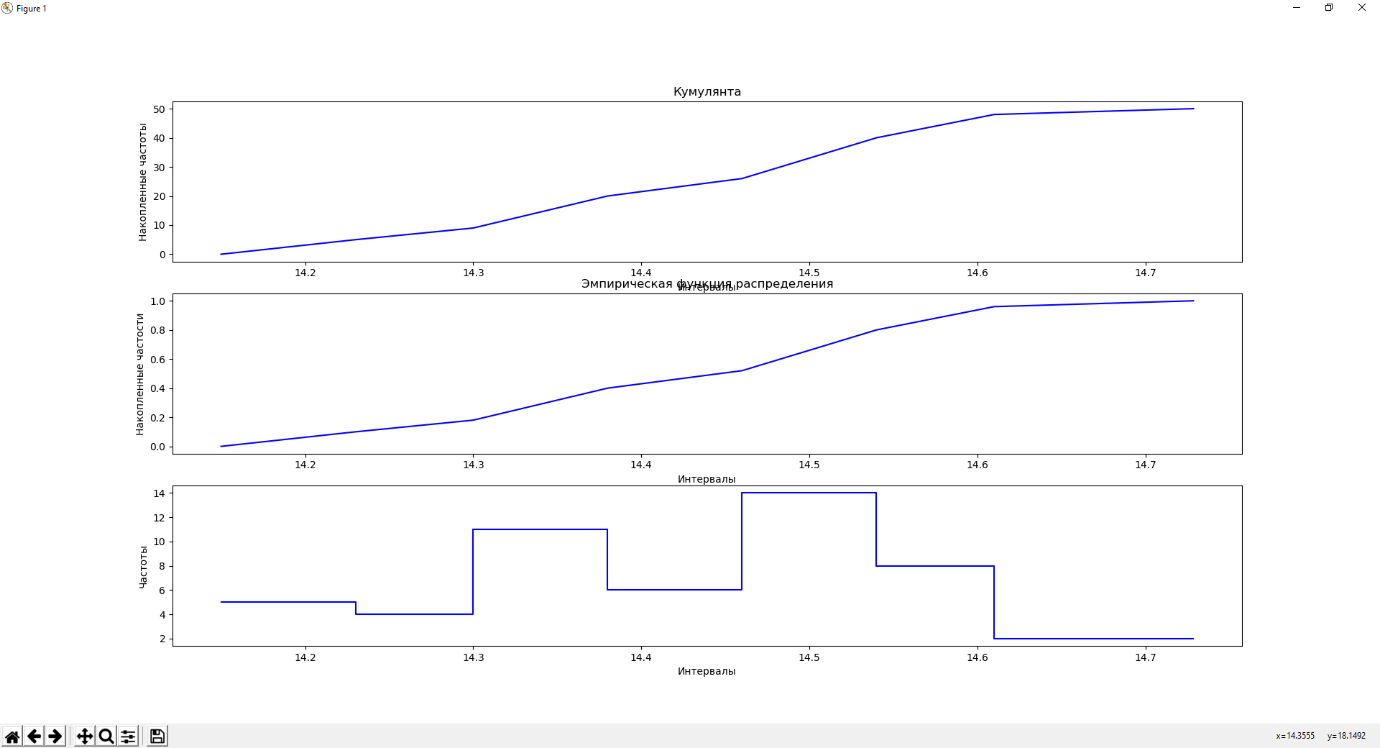
# Задача №4

Постановка задачи: Измерения диаметров 50 валиков, выточенных на станке, дали следующие

результаты (в мм): 14,51 14,42 14,56 14,47 14,46 14,35 14,48 14,53 14,21 14,31 14,35 14,68 14,56 14,28 14,36 14,21 14,52 14,23 14,41 14,46 14,69 14,54 14,36 14,15 14,37 14,51 14,25 14,55 14,51 14,36 14,62 14,55 14,38 14,33 14,40 14,52 14,48 14,51 14,55 14,39 14,54 14,58 14,48 14,37 14,38 14,51 14,36 14,15 14,24 14,32

Построить интервальный вариационный ряд и графически отобразить

Результаты:



Код программы:

def task4():  
 values = [  
 14.51, 14.42, 14.56, 14.47, 14.46, 14.35, 14.48, 14.53,  
 14.21, 14.31, 14.35, 14.68, 14.56, 14.28, 14.36, 14.21,  
 14.52, 14.23, 14.41, 14.46, 14.69, 14.54, 14.36, 14.15,  
 14.37, 14.51, 14.25, 14.55, 14.51, 14.36, 14.62, 14.55,  
 14.38, 14.33, 14.40, 14.52, 14.48, 14.51, 14.55, 14.39,  
 14.54, 14.58, 14.48, 14.37, 14.38, 14.51, 14.36, 14.15,  
 14.24, 14.32  
 ]  
 v = ContinuousVS(values)  
 v.draw\_cumulate()\  
 .draw\_empiric\_dist\_func()\  
 .draw\_hist()

**Deep dark back end**

import collections

import math

import abc

import plotly.graph\_objects as go

# Common variation series calculations

class VariationSeries:

    """Abstacrt"""

    def \_\_init\_\_(self, values: [dict, list], precision=5):

        self.precision = precision

        # User passes list of raw values

        if isinstance(values, list):

            self.vals = sorted(values)

            self.vs = self.build\_vs\_from\_list(values)

        # User passes dict with a ready variation series

        elif isinstance(values, dict):

            self.vs = self.build\_vs\_from\_dict(values)

            self.vals = []

            for x, n in self.vs.items():

                self.vals.extend([x]\*n)

    def draw\_cumulate(

            self, title='Кумулянта (заголовок)', name='Кумулянта (легенда)',

            x\_label='Варианты', y\_label='Накопленные частоты'

        ):

        """Draw cumulate: variants..closest\_x => accumulated\_frequencies"""

        x = self.variants + [self.closest\_x]

        y = self.acc\_frequencies

        VariationSeries.\_prepare\_figure(x, y, title, name, x\_label, y\_label).show()

        return self

    def draw\_empiric\_dist\_func(

            self, title='Эмпирическая функция распределения (заголовок)',

            name='Эмпирическая функция распределения (легенда)',

            x\_label='Варианты', y\_label='Частости'

        ):

        """Draw empiric distribution function: variants..closest\_x => accumulated\_relative\_frequencies"""

        x = self.variants + [self.closest\_x]

        y = self.acc\_rel\_frequencies

        VariationSeries.\_prepare\_figure(x, y, title, name, x\_label, y\_label).show()

        return self

    @property

    def n(self) -> int:

        """Number of elements in the variation series"""

        return len(self.vals)

    @property

    def x\_max(self) -> [float, int]:

        if isinstance(self.vals[0], tuple):

            return self.vals[-1][1]

        return max(self.vals)

    @property

    def x\_min(self) -> [float, int]:

        if isinstance(self.vals[0], tuple):

            return self.vals[0][0]

        return min(self.vals)

    # Different for discrete and continuous vs

    @property

    @abc.abstractmethod

    def variants(self) -> list:

        """Variants x"""

        pass

    @property

    def acc\_frequencies(self) -> list:

        """Accumulated frequencies m(x(i))"""

        def gen\_acc\_frequencies(values):

            """Generator of accumulated sum"""

            accumulated\_sum = 0

            for m\_i in values:

                yield accumulated\_sum

                accumulated\_sum += m\_i

            yield accumulated\_sum

        return list(gen\_acc\_frequencies(self.var\_frequencies))

    @property

    def acc\_rel\_frequencies(self) -> list:

        """Accumulated relative frequencies w(x)"""

        n = self.n

        p = self.precision

        return [round(m\_x / n, p) for m\_x in self.acc\_frequencies]

    @property

    def var\_frequencies(self) -> list:

        """Variants frequencies m(i)"""

        return list(self.vs.values())

    @property

    def median(self) -> [int, float]:

        """Return median of the variation series"""

        n = self.n

        idx = n // 2

        if n % 2 == 0:

            idx -= 1

        return self.vals[idx]

    @property

    def mode(self) -> [int, float]:

        """Return mode of the variation series"""

        items = list(self.vs.items())

        mode\_val, mode\_reps = items[0]

        for val, reps in items:

            if mode\_reps < reps:

                mode\_val = val

                mode\_reps = reps

        return mode\_val

    @property

    @abc.abstractmethod

    def closest\_x(self) -> [int, float]:

        """Return closests x value for accumulated functions"""

        pass

    @abc.abstractmethod

    def build\_vs\_from\_list(self, values: list) -> 'collections.OrderedDict':

        """Build variation series from raw values"""

        pass

    @staticmethod

    def build\_vs\_from\_dict(values: dict) -> 'collections.OrderedDict':

        """Build variation series from dict"""

        # User shoud be consient about values he passes

        vs = collections.OrderedDict(sorted(values.items()))

        return vs

    @staticmethod

    def \_prepare\_figure(x, y, title, name, x\_label, y\_label) -> 'go.Figure':

        fig = go.Figure()

        scatter = go.Scatter(x=x, y=y, mode='lines', name=name)

        fig.add\_trace(scatter)

        fig.update\_layout(title=title, xaxis\_title=x\_label, yaxis\_title=y\_label)

        return fig

# Discrete variation series

class DiscreteVS(VariationSeries):

    def draw\_polygon(

        self, title='Полигон (заголовок)', name='Полигон (легенда)',

        x\_label='Варианты', y\_label='Частоты'

        ):

        """Draw polygon: variants => variants frequencies"""

        x = self.variants

        y = self.var\_frequencies

        VariationSeries.\_prepare\_figure(x, y, title, name, x\_label, y\_label).show()

        return self

    @property

    def variants(self) -> list:

        return list(self.vs.keys())

    @property

    def closest\_x(self) -> [float, int]:

        """The closests x for accumulated values"""

        variants = self.variants

        penult, last = variants[-2:]

        diff = last - penult

        return last + diff

    def build\_vs\_from\_list(self, values: list) -> 'collections.OrderedDict':

        vs = collections.OrderedDict()

        for val in sorted(values):

            vs[val] = vs.get(val, 0) + 1

        return vs

# Continuous variation series

class ContinuousVS(VariationSeries):

    def draw\_hist(self, title='Гистограмма'):

        x = self.vals

        if isinstance(x[0], tuple):

            x = [x[0][0]] + [right for \_, right in x]

        xbins = {

            'start': self.x\_min, 'end': self.x\_max,

            'size': self.delta,

        }

        trace = go.Histogram(x=x, xbins=xbins, autobinx=False)

        fig = go.Figure()

        fig.add\_trace(trace)

        fig.show()

        return self

    @property

    def k(self) -> int:

        """Number of intervals"""

        return math.ceil(1 + 1.4\*math.log(self.n))

    @property

    def delta(self) -> [float, int]:

        """Interval length"""

        return (self.x\_max - self.x\_min) / self.k

    @property

    def intervals(self) -> list:

        return list(self.vs.keys())

    @property

    def variants(self) -> list:

        return [left for left, right in self.intervals]

    @property

    def closest\_x(self) -> [int, float]:

        return self.intervals[-1][1]

    def build\_vs\_from\_list(self, values: list) -> 'collections.OrderedDict':

        """Build variation series from raw values"""

        # There are raw values

        p = self.precision

        delta = self.delta

        x\_min = self.x\_min

        x\_max = self.x\_max + delta/2

        intervals = []

        for i in range(self.k):

            bias = i \* delta

            left = round(x\_min + bias, p)

            right = round(x\_min + bias + delta, p)

            intervals += [(left, right)]

        # Move the last interval for capturing

        last\_interval = intervals[-1]

        new\_last\_interval = (last\_interval[0], x\_max)

        intervals[-1] = new\_last\_interval

        vs = collections.OrderedDict()  # Frequncies

        for val in sorted(self.vals):

            # Choose the correct interval

            for interval in intervals:

                left, right = interval

                if left <= val < right or val == x\_max:

                    vs[interval] = vs.get(interval, 0) + 1

                    break

        return vs