# 1. ЛАБОРАТОРНАЯ РАБОТА № 1
# «ИССЛЕДОВАНИЕ БАЗОВЫХ ФУНКЦИЙ ЯЗЫКА PYTHON»

## 1.1. Цель работы

Изучение технологии подготовки и выполнения программ на языке Python, исследование свойств функций языка Python, используемых при обработке последовательностей, формирование навыков определения классов языка Python

## 1.2. Краткие теоретические сведения

### 1.2.1. Python

Python — динамически типизированный, мультипарадигменный язык программирования, официально опубликованный в 1991 году. Разработан Гвидо ван Россумом (Guido van Rossum) в Национальном исследовательском институте математики и компьютерных наук (г. Амстердам).

Python быстро стал одним из самых популярных языков программирования. В настоящее время является одним из основных языков обработки данных и искусственного интеллекта.

### 1.2.2 Установка дистрибутива Anaconda и загрузка задания

В лабораторных работах рекомендуется использовать дистрибутив Anaconda Python, отличающийся простотой установки. В него входит практически всё необходимое для выполнения лабораторных работ, в том числе:

- интерпретатор IPython;
- большинство библиотек Python;
- локальный сервер Jupyter Notebook для загрузки и выполнения блокнотов;
- интегрированная среда разработки Spyder IDE (IDE - Integrated Development Environment).

Все задания из лабораторных работ протестированы в версии Python 3.9.

Программу установки Python с помощью Anaconda для Windows, macOS и Linux можно загрузить по адресу: https://www.anaconda.com/download/

Когда загрузка завершится, запустите программу установки. Чтобы установленная копия Anaconda работала правильно, не перемещайте ее файлы после установки.

Если в вашей системе уже была установлена иная версия Python, то рекомендуется создать отдельную среду для работы с Python 3.9 (можно использовать версии Python 3.9-3.11), введя в командном окне conda команду:

```
conda create --name <env-name> python=3.9
```

Здесь <env-name> - название среды, например, python39. Чтобы войти в среду, которая была только что создана, активируйте её:

```
conda activate python39
```

Проверьте используемую в среде версию Python, набрав в команду:

```
python –V
   Python 3.9.7
```

Установите в созданную среду Spyder IDE командой: conda install spyder. Также установите библиотеки для работы с динамическими массивами numpy и визуализации данных matplotlib:

```
conda install numpy
conda install matplotlib
```

Среда готова для работы. Для выхода из среды python39 выполните команду деактивации

```
conda deactivate
```

Вы вернетесь к версии Python, которая была ранее установлена в системе.

Все файлы, которые содержат задания данной лабораторной работы, находятся архиве: **МиСИИ_лаб1_2024.zip**. Файл можно получить у преподавателя или найти по адресу размещения электронных ресурсов дисциплины в системе Moodle. Создайте локальную рабочую папку, откройте её и загрузите все файлы из указанного архива

### 1.2.3. Вызов интерпретатора Python

Python можно запускать в одном из двух режимов. Его можно использовать интерактивно, работая с интерпретатором, или вызвать из командной строки для выполнения сценария.

Покажем, как использовать интерпретатор Python в интерактивном режиме. Откройте окно командной строки в своей системе и запустите **командную строку Anaconda** из меню Пуск. В окне командной строки введите команду **python** и нажмите Enter. На экране появится сообщение, подобное изображенноиу на рисунке 1.1. (зависит от платформы и версии Python):

```
(base) C:\Users\User>python
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc
. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Рисунок 1.1 – Экранное сообщение при выполнении команды python

Строка с символами ">>>" на рисунке1.1 обозначает *приглашение*, означающее, что Python ожидает ввода. Чтобы выйти из интерактивного режима введите команду exit().

Вы также можете работать в интерактивном режиме, используя непосредственно оболочку IPython. Для этого вместо вышеупомянутой команды python следует ввести команду ipython. На экране появится сообщение, подобное изображенному на рисунке 1.1, но строка приглашения теперь будет в виде “In[1]”.

Среди функций, которые предоставляет IPython, наиболее интересны следующие:

динамический анализ объектов;

доступ к оболочке системы через командную строку;

прямая поддержка профилирования;

работа с отладочными средствами.

IPython является частью более крупного проекта под названием Jupyter, предоставляющего возможность работы с интерактивными блокнотами через интернет-браузер.

Также возможна работа с IPython в интегрированной среде Spyder IDE, которая предоставляет традиционный спектр возможностей, свойственных IDE. При этом консольное окно IPython обычно располагается внизу справа основного экрана Spyder IDE (рисунок 1.2).

Ниже будут приведены примеры команд для консоли Python. Команды консоли IPython аналогичны, но требуют ввода команд, исполняемых операционной средой, начиная со знака «!», например:

```
!python -V
```

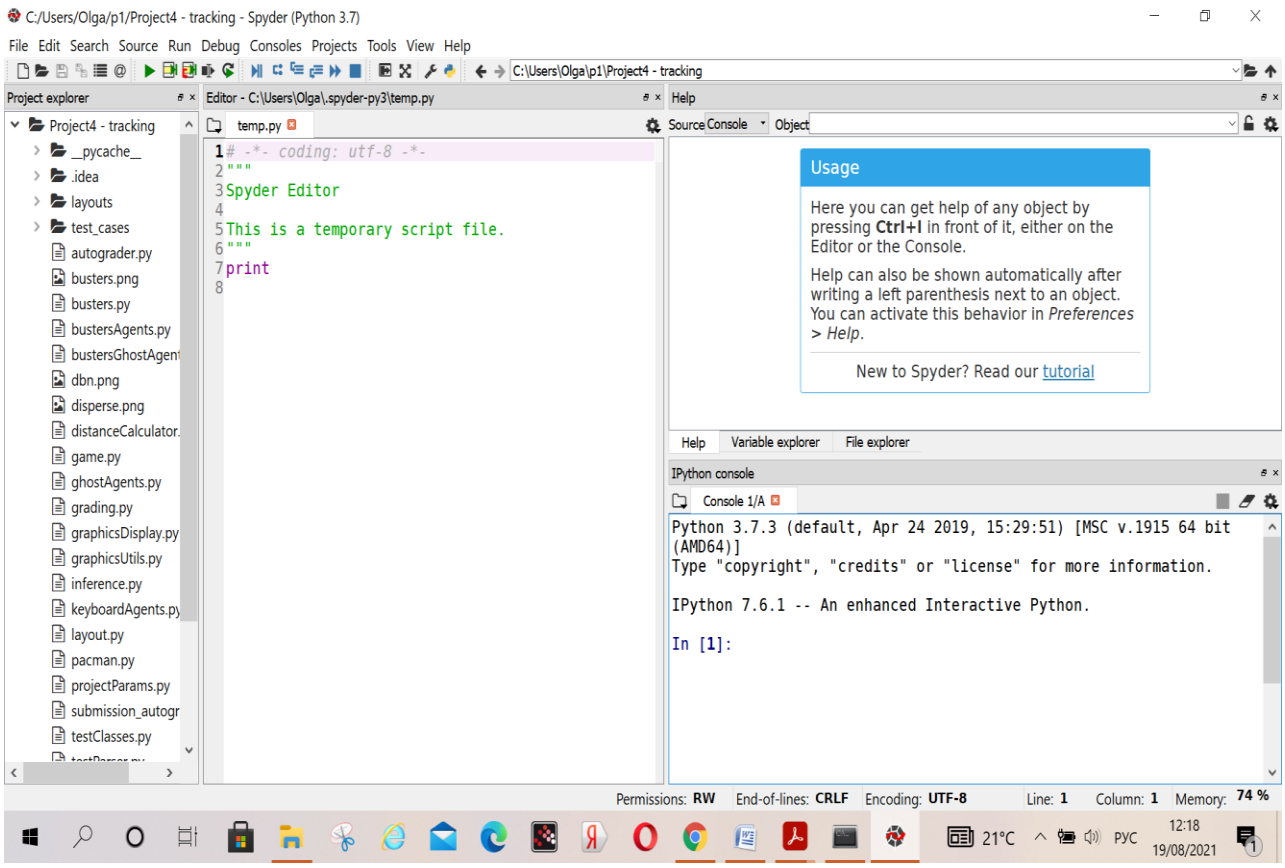


Рисунок 1.2 – Основной экран интегрированной среды Spyder IDE

### 1.2.4. Введение в программирование на языке Python

#### Арифметические и логические операторы

Интерпретатор Python можно использовать для вычисления значений арифметических выражений. Если вы введёте арифметическое выражение после приглашения ">>>", то оно будет вычислено, а результат будет возвращен в следующей строке:

```
>>>2+3
5
>>>2*3
6
>>> num = 8.0                # присваивание
>>> num + = 2.5              # инкремент
>>> print (num)              # печать
10.5
```

Python поддерживает работу с различными типами числовых данных: int, float, complex. Чтобы получить информацию о типе значения переменной, воспользуйтесь встроенной *функцией* Python type:

```
>>> type(num)
float
```

В таблице 1.1 перечислены арифметические операторы Python.

Таблица 1.1 - Арифметические операторы Python

| Операция Python | Арифметический оператор | Алгебраическое выражение | Выражение Python |
|---|---|---|---|
| Сложение | + | $f + 7$ | `f + 7` |
| Вычитание | – | $p - c$ | `p - c` |
| Умножение | * | $b \cdot m$ | `b * m` |
| Возведение в степень | ** | $x^y$ | `x ** y` |
| Деление | / | $x/y$, или $\frac{x}{y}$, или $x \div y$ | `x / y` |
| Целочисленное деление | // | $[x/y]$, или $\left\lfloor \frac{x}{y} \right\rfloor$, или $[x \div y]$ | `x // y` |
| Остаток от деления | % | $r \bmod s$ | `r % s` |

В Python также существуют логические операторы и операции отношений, выполнящие или возвращающие результаты действий с логическими значениями типа bool (True и False) :

```
>>>1==0                    # равно
False
>>>1!=0                    # не равно
True
>>> not (1==0)            # логическое отрицание
True
>>> (2==2) and (2==3)     # логическое И
False
>>> (2==2) or (2==3)      # логическое ИЛИ
True
```

**Строки**

Python имеет встроенный строковый тип (класс) str. Сроковая константа заключается в одинарные или двойные кавычки: ‘hello’, “hello”. Оператор “+” позволяет выполнить конкатенацию строк:

```
>>>’искусственный’+”интеллект”
’искусственныйинтеллект’
```

Существует много полезных встроенных методов для работы со строками, например:

```
>>> 'artificial'.upper ()
'ARTIFICIAL'
>>> 'HELP'.lower ()
'help'
>>> len('Help')
4
```

Строки являются *неизменяемыми* — это означает, что их нельзя модифицировать в программе. Вы можете прочитать отдельные символы в строке, но при попытке присвоить новое значение одному из символов строки будет ошибка TypeError:

```
>>> s = 'hello'
>>> s[0]
'h'
s[0] = 'H'
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-15-812ef2514689> in <module>()
----> 1 s[0] = 'H'
TypeError: 'str' object does not support item assignment
```

Python позволяет сохранять строки (строковые выражения) в переменных:

```
>>> hw='%s %s %d' % ('hello', 'world', 11)          #форматирование вывода
```

```
>>> print(hw)
hello world 11
>>> hw.replace('l','ell')                    #замена символов
'heellello worelld 11'
```

Если в строке встречается обратный слеш (\), то он является *управляющим символом*, а в сочетании с непосредственно следующим за ним символом образует *управляющую последовательность* (\n, \t , \\, \',\"):

```
>>> print('Welcome\nto\n\nPython!')       # \n – управляющий символ новой строки
Welcome
to

Python!
```

В Python имеются строки в тройных кавычках. Используйте их для создания:
многострочных строк;
строк, содержащих одинарные или двойные кавычки;
*doc-строк* — рекомендуемого способа документирования некоторых компонентов программы. Например:

```
>>> print("""Display "hi" and 'bye' in quotes""")
Display "hi" and 'bye' in quotes
```

Python предоставляет возможности форматирования значений с использованием форматных строк (*f - строк*).

```
>>> average = 15.6666
>>> print(f'Среднее равно {average:.2f}')
Среднее равно 15.67
```

Буква f перед открывающей кавычкой строки означает, что это форматная строка. Чтобы указать, где должно вставляться выводимое значение, используйте поля в фигурных скобках { }. Поле с переменной {average} преобразует значение переменной average в строковое представление, а затем меняет {average} *заменяющим текстом*. В форматной строке за переменной (выражением) в заполнителе может следовать двоеточие (:) и *форматный спецификатор*, который описывает, как должен форматироваться заменяющий текст. Форматный спецификатор .2f форматирует average при выводе как число с плавающей точкой (f) с двумя цифрами в дробной части (.2). Выражения в фигурных скобках (в поле) могут содержать значения, переменные или другие выражения (например, вычисления или вызовы функций).

В приведенном примере спецификатор ‘.2f’ определял тип представления переменной. Существуют и другие типы представлений (b, o, x и X), форматирующие целые числа для представления в двоичной, восьмеричной и шестнадцатеричной системах счисления.

В f-строках можно задавать ширину поля вывода и выполнять выравнивание. Например:

```
>>> f'[{3.5:<15f}]'                  # выравнивание слева в поле из 15 позиций
'[3.500000       ]'
>>> f'[{"hello":>15}]'               # выравнивание справа в поле из 15 позиций
'[          hello]'
```

Форматные строки Python были добавлены в язык в версии 3.6. До этого форматирование строк выполнялось с помощью метода format. Собственно, функциональность форматных строк основана на средствах метода format. Метод format может вызываться для *форматной строки*, содержащей *заполнители* в фигурных скобках { }:

```
'{:.2f}'.format(14.989)
'14.99'
```

. Чтобы узнать, какие методы Python предоставляет для работы с тем или иным типом данных, используйте команды dir и help. Например, методы для работы со строками:

```
>>>s=’abc’
>>>dir(s)
```

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

```
>>>help(s.find)

Help on built-in function find:

find(...) method of builtins.str instance
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

Функция find возвращает наименьший индекс s, начиная с которого в s найдена подстрока sub. Необязательные аргументы start и end интерпретируются как обозначения сегмента строки для поиска:

```
>>>s.find('b')
1
```

**Контейнеры: списки, кортежи, словари, множества**

Python оснащен некоторыми полезными встроенными структурами данных – списками, кортежами, словарями, множествами.

**Списки**

В списках (класс list) хранится последовательность изменяемых элементов:

```
>>> fruits = ['apple', 'orange', 'pear', 'banana']   # задание списка
>>> fruits[0]                                        # обращение по индексу
'apple'
```

Можно создать список из целых чисел с помощью функций range() и list():

```
>>> nums=list(range(5))     #range формирует последовательность от 0 до 4
>>> print(nums)
[0, 1, 2, 3, 4]
```

Здесь вызов функции range(5) создает итерируемый объект, который представляет последовательность целых чисел от 0 и до значения аргумента 5, *не включая* последний, в данном случае 0, 1, 2, 3, 4. Функция list создает из последовательности чисел список [0, 1, 2, 3, 4]

Для *объединения* списков можно использовать оператор сложения (+):

```
>>> otherFruits = ['kiwi', 'strawberry']
>>> fruits + otherFruits
>>> ['apple', 'orange', 'pear', 'banana', 'kiwi', 'strawberry']
```

*Обращение* к элементам списков выполняется с помощью индексов в квадратных скобках: fruits[0] – первый элемент списка. Python также допускает отрицательную индексацию с конца списка. Например, fruits[-1] обращается к последнему элементу 'banana', а

```
>>> fruits[-2]                              #2-ой с конца
'pear'
```

вернет второй элемент списка с его конца.

Можно *индексировать несколько смежных элементов* одновременно с помощью операторов среза (сегментирования). Например, fruits[1:3] возвращает список, содержащий элементы в позициях 1 и 2. В общем, fruits[start: stop] вернет элементы в позициях start, start + 1, ..., stop-1. Обращение fruits[start:] вернет все элементы списка, начиная с индекса start. Также fruits[:end] вернет все элементы с начала списка и до элемента в позиции end. Примеры:

```
>>> fruits=['apple', 'orange', 'pear', ' banana']
>>> fruits[0:2]
['apple', 'orange']
>>> fruits[:3]
['apple', 'orange', 'pear']
>>> fruits[2:]
['pear', 'banana']
>>> len(fruits)
4
```

Для работы со списками в Python имеется большой набор встроенных методов. Например,

```
>>> fruits.pop()                          #удаляет последний эл-т и возвращает его
'banana'
>>> fruits
['apple', 'orange', 'pear']                #состояние списка после pop
>>> fruits.append('grapefruit')           # добавляет в конец списка
>>> fruits
['apple', 'orange', 'pear', 'grapefruit']  #состояние списка после добавления
>>> fruits[-1] = 'pineapple'              #заменить последний эл-т
>>> fruits
['apple', 'orange', 'pear', 'pineapple']   #состояние после замены
>>> fruits. insert(0, ‘grapefruit’)       #вставка эл-та в заданную позицию
>>> fruits
['grapefruit', 'apple', 'orange', 'pear', 'pineapple']
```

Элементы, хранящиеся в списках, могут быть любого типа. Так, например, элементами списка могут быть списки:

```
>>> lstOfLsts = [['a', 'b', 'c'], [1, 2, 3], ['one', 'two', 'three']]
>>> lstOfLsts[1][2]
3
>>> lstOfLsts[0].pop()
'c'
>>> lstOfLsts
[['a', 'b'], [1, 2, 3], ['one', 'two', 'three']]
```

*Команда* del может использоваться для удаления элементов из списка. Вы можете удалить элемент с любым действующим индексом или элемент(-ы) любого допустимого сегмента. Создадим список, а затем воспользуемся del для удаления его последнего элемента:

```
>>> numbers = list(range(0, 10))          # range генерирует числа от 0 до 9
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del numbers[-1]
numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Следующий пример удаляет из списка сегмент из первых двух элементов:

```
>>> del numbers[0:2]
>>> numbers
[2, 3, 4, 5, 6, 7, 8]
```

Метод списков sort *изменяет* список и сортирует элементы по возрастанию:

```
>>> numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
>>> numbers.sort()
>>> numbers
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Чтобы отсортировать список по убыванию, вызовите sort с необязательным ключевым аргументом reverse, равным True (по умолчанию используется значение False):

```
>>> numbers.sort(reverse=True)
>>> numbers
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Встроенная функция sorted *возвращает новый список*, содержащий отсортированные элементы своей *последовательности*-аргумента — исходная последовательность при этом *не изменяется*.

```
>>> numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
>>> ascending_numbers = sorted(numbers)
>>> ascending_numbers
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers
[10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
```

Необязательный ключевой аргумент reverse со значением True заставляет функцию отсортировать элементы по убыванию.

Часто бывает нужно определить, содержит ли список (последовательность) значение, соответствующее заданному *ключу поиска*. У списков имеется метод index, которому передается ключ поиска в качестве аргумента. Метод начинает поиск с индекса 0 и возвращает индекс *первого* элемента, который равен ключу поиска:

```
>>> numbers = [3, 7, 1, 4, 2, 8, 5, 6]
>>>  numbers.index(5)                  # 5 – ключ поиска
```

```
6
```

Если искомое значение отсутствует в списке, то происходит ошибка ValueError. Код ниже ищет в списке значение 5, начиная с индекса 7 и до конца списка:

```
>>> numbers = [3, 7, 1, 4, 2, 8, 5, 6, 3, 7, 1, 4, 2, 8, 5, 6]
>>> numbers.index(5, 7)
14
```

Оператор in проверяет, содержит ли итерируемый объект, каковым является список, заданное значение:

```
>>> 1000 in numbers
False
>>> 5 in numbers
True
```

### Кортежи

Кортеж (класс tuple) – упорядоченный список значений, отличающийся от списка тем, что может быть использован в качестве ключа в словаре и в качестве элемента множества (см. далее). Обратите внимание, что кортежи заключаются в круглые скобки, а списки - в квадратные скобки.

Чтобы создать пустой кортеж, используйте пустые круглые скобки:

```
>>> student_tuple = ()
>>> student_tuple
()
>>> len(student_tuple)
0
```

Для упаковки элементов в кортеж можно перечислить их, разделяя запятыми:

```
>>> student_tuple = 'John', 'Green', 3.3  #создание кортежа
>>> student_tuple
('John', 'Green', 3.3)
>>> len(student_tuple)
3
>>> student_tupl[0]                       #обращение по индексу
'John'
```

Когда Python выводит кортеж, он всегда отображает его содержимое в круглых скобках. Список значений кортежа, разделяемых запятыми, также можно при создании заключать в круглые скобки (хотя это и не обязательно):

```
>>> pair = (3, 5)          #упаковка значений в кортеж
>>> x, y = pair            #распаковка (извлечение) кортежа
>>> x
3
```

```
>>> y
5
```

Кортеж похож на список, за исключением того, что вы не можете изменить его после создания:

```
>>>pair[1] = 6                #попытка изменения значения элемента кортежа
TypeError: object does not support item assignment
```

Попытка изменить неизменяемую структуру вызывает исключение. Исключения указывают на ошибки, например, выход индекса за границы, ошибки типа и т. п. Тем не менее, можно добавлять элементы в кортеж, кортежи могут содержать изменяемые объекты, кортежи можно добавлять в списки. Пусть

```
>>> tuple1 = (10, 20, 30)
>>> tuple2 = tuple1          # tuple 2 и tulple 1 ссылаются на один и тот же кортеж
>>> tuple2
(10, 20, 30)
```

При конкатенации кортежа (40, 50) с tuple1 создается *новый* кортеж, ссылка на который присваивается переменной tuple1, тогда как tuple2 все еще ссылается на исходный кортеж:

```
# добавление элементов в кортеж
>>> tuple1 += (40, 50)       # справа от += должен быть кортеж
>>> tuple1
(10, 20, 30, 40, 50)
>>> tuple2
(10, 20, 30)
```

Конструкция += также может использоваться для присоединения кортежа к списку:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> numbers += (6, 7)
>>> numbers
[1, 2, 3, 4, 5, 6, 7]
```

Кортежи могут содержать изменяемые объекты. Создадим кортеж student_tuple с именем, фамилией и списком оценок:

```
>>> student_tuple = ('Amanda', 'Blue', [98, 75, 87])
```

И хотя сам кортеж неизменяем, его элемент-список может изменяться:

```
>>>  student_tuple[2][1] = 85
>>> student_tuple
('Amanda', 'Blue', [98, 85, 87])
```

### Множества

Множество – это еще одна структура данных, которая является неупорядоченным списком без повторяющихся элементов. Множество можно создать из списка:

```
>>> shapes = ['circle', 'square', 'triangle', 'circle']
>>> setOfShapes = set(shapes)
```

Ниже представлены примеры, как добавлять элементы во множество, проверять, входит ли элемент во множество, и выполнять операции над множествами (разность, пересечение, объединение):

```
>>> setOfShapes
set(['circle', 'square', 'triangle'])
>>> setOfShapes.add('polygon')                    #добавление
>>> setOfShapes
set(['circle', 'square', 'triangle', 'polygon'])
>>> 'circle' in setOfShapes                       #принадлежность
True
>>> 'rhombus' in setOfShapes
False
>>> favoriteShapes = ['circle', 'triangle', 'hexagon']
>>> setOfFavoriteShapes = set(favoriteShapes)
>>> setOfShapes – setOfFavoriteShapes                  #разность
set(['square', 'polygon'])
>>> setOfShapes & setOfFavoriteShapes                  #пересечение
set(['circle', 'triangle'])
>>> setOfShapes | setOfFavoriteShapes                  #объединение
set(['circle', 'square', 'triangle', 'polygon', 'hexagon'])
```

Обратите внимание на то, что множества неупорядочены.

### Словари

Словарь (класс dict) – это структура, которая обеспечивает отображение одного объекта (ключ) на другой объект (значение). Словари также называют ассоциативными списками, так как они содержат пары *ключ-значение*, заключаемые в фигурные скобки. Ключ должен быть неизменяемым типом (строка, число или кортеж). Значение может быть любым типом данных Python.

В приведенном ниже примере порядок печати ключей, возвращаемых Python, может отличаться от показанного ниже. Причина в том, что в отличие от списков с фиксированным порядком, словарь - это просто хеш-таблица, для которой нет фиксированного порядка ключей. Порядок следования ключей зависит от того, как

именно алгоритм хеширования сопоставляет ключи с сегментами, и обычно является произвольным. При кодировании Вы не должны полагаться на порядок следования ключей. Примеры основных операций со словарями:

```
>>> studentIds = {'knuth': 42.0, 'turing': 56.0, 'nash': 92.0}      #создание
>>> studentIds['turing']                                            #поиск по ключу
56.0
>>> studentIds['nash'] = 'ninety-two'                               #замена значения
>>> studentIds
{'knuth': 42.0, 'turing': 56.0, 'nash': 'ninety-two'}
>>> del studentIds['knuth']                                  #удаление
>>> studentIds
{'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds['knuth'] = [42.0, 'forty-two']                       #значение список
>>> studentIds
{'knuth': [42.0, 'forty-two'], 'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds.keys()                                               #список ключей
['knuth', 'turing', 'nash']
>>> studentIds.values()                                             # список значений
[[42.0, 'forty-two'], 56.0, 'ninety-two']
>>> studentIds.items()                                              #список пар
[('knuth', [42.0, 'forty-two']), ('turing',56.0), ('nash', 'ninety-two')]
>>> len(studentIds)
3
```

Обратите внимание, что метод items возвращает пары *ключ-значение* в форме кортежей.

### 1.2.5 Написание скриптов

Написание скриптов требует знакомства с управляющими конструкциями Python (if и else, for, while и др.). Так как они подобны управляющим конструкциям других языков программирования, то просто покажем их использование на примерах. Детальнее с ними можно ознакомиться в соответствующем разделе официального руководства Python по адресу: https://docs.python.org/3.6/tutorial/

При написании скриптов будьте внимательны с отступами. В отличие от многих других языков, Python для структурирования программного кода и его корректной интерпретации использует отступы. Так, например, следующий скрипт:

```
if 0 == 1:
    print('We are in a world of sport')
print('Thank you for playing')
```

выведет фразу “Thank you for playing”. Но если мы его перепишем так:

```
if 0 == 1:
    print('We are in a world of sport')
    print('Thank you for playing')
```

то ничего выводиться не будет. Обычно используется отступ в 4 позиции.

Теперь, когда вы научились использовать Python в интерактивном режиме, давайте напишем простой скрипт на Python, демонстрирующий использование цикла for. Откройте файл с именем foreach.py, который содержит следующий код:

```
# цикл по элементам списка fruits
fruits = ['apples', 'oranges', 'pears', 'bananas']
for fruit in fruits:
    print(fruit + ' for sale')

# цикл по парам “ключ-значение”из словаря fruitPrices
fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}
for fruit, price in fruitPrices.items():
    if price < 2.00:
        print('%s cost %f a pound' % (fruit, price))
    else:
        print(fruit + ' are too expensive!')
```

Чтобы выполнить скрипт, в командной строке введите команду (предварительно командой cd перейдите в папку с файлом foreach.py):

```
C:\user\user\ai\lab1> python foreach.py
```

В результате выполнения программы получим:

```
apples for sale
oranges for sale
pears for sale
bananas for sale
apples are too expensive!
oranges cost 1.500000 a pound
pears cost 1.750000 a pound
```

Помните, что результаты печати словаря с ценами могут располагаться в другом порядке, чем в показано.

Циклический просмотр элементов списка и их индексов можно организовать с использованием цикла for и функции enumerate:

```
fruits = ['apples', 'oranges', 'pears', 'bananas']
for idx, fruit in enumerate(fruits):   #enumerate возвращает (idx,fruit)
    print(‘#%d:%s’ % (idx, fruit))
```

В результате выполнения этого кода получим:

```
#0:apples
#1:oranges
#2:pears
#3:bananas
```

Часто при создании новых списков используют конструкцию *списковое включение* (list comprehensions), которая обеспечивает компактную замену конструкции с for. Например, следующий код

```
list1 = [ ]
for item in range(1, 6):       # range формирует последовательность от 1 до 5
   list1.append(item)
```

создает список [1, 2, 3, 4, 5]. Этот же список можно создать с помощью спискового включения:

```
list1=[x for x in range(1,6)]
```

Списковое включение позволяет выполнять разные операции (например, вычисления), *отображающие* элементы на новые значения (в том числе, возможно, и других типов). Отображение (mapping) — стандартная операция функционального программирования, которая возвращает результат с *тем же* количеством элементов, что и в исходных отображаемых данных [4, 5].

Другая распространенная операция программирования в функциональном стиле — *фильтрация* элементов и отбор только тех элементов, которые удовлетворяют заданному условию. Как правило, при этом строится список с *меньшим* количеством элементов, чем в фильтруемых данных. Чтобы выполнить эту операцию с использованием спискового включения, используйте *секцию* if. Следующий фрагмент кода демонстрирует простое отображение и фильтрацию с помощью *спискового включения*:

```
#отображение nums на plusOneNums
nums = [1, 2, 3, 4, 5, 6]
plusOneNums = [x + 1 for x in nums]

#фильтрация списков
# создаем список из нечетных элементов nums
oddNums = [x for x in nums if x % 2 == 1]
print(oddNums)

# создаем список из нечетных элементов plusOneNums
oddNumsPlusOne = [x + 1 for x in nums if x % 2 == 1]
print(oddNumsPlusOne)
```

Этот код находится в файле с именем listcomp.py, который вы можете запустить командой:

```
C:\user\user\ai\lab1> python listcomp.py
[1, 3, 5]
[2, 4, 6]
```

### 1.2.6 Выражения-генераторы

*Выражение-генератор* отчасти напоминает списковое включение, но оно создает итерируемый *объект-генератор*, производящий значения *по требованию* [5]. Этот механизм называется *отложенным вычислением*. В списковом включении используется *быстрое вычисление,* позволяющее создавать списки в момент выполнения. При большом количестве элементов создание списка может потребовать значительных объёмов памяти и затрат времени. Таким образом, выражения-генераторы могут сократить потребление памяти программой и повысить быстродействие, если не все содержимое списка понадобится одновременно.

Выражения-генераторы обладают теми же возможностями, что и списковое включение, но они определяются в круглых скобках вместо квадратных. Ниже выражение-генератор возвращает квадраты только нечетных чисел из numbers:

```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
for value in (x ** 2 for x in numbers if x % 2 != 0):
        print(value, end=' ')
```

В результате выполнения этого кода будут выведены числа: 9 49 1 81 25.

Чтобы показать, что выражение-генератор не создает список, присвоим выражение-генератор из предыдущего фрагмента переменной и выведем значение этой переменной:

```
>>> squares_of_odds = (x ** 2 for x in numbers if x % 2 != 0)
>>> squares_of_odds
<generator object <genexpr> at 0x1085e84c0>
```

Текст "generator object <genexpr>" сообщает, что square_of_odds является объектом-генератором, который был создан на базе выражения-генератора (объект genexpr).

### 1.2.7 Определение функций

Функция в языке Python определяются с использованием заголовка def *имя функции* (*параметры*). Ниже приведен пример определения и вызова функции, которая выводит стоимость покупки фруктов:

```
fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}

# Определение функции buyFruit
def buyFruit(fruit, numPounds):
    if fruit not in fruitPrices:
        print("Sorry we don't have %s" % (fruit))
    else:
        cost = fruitPrices[fruit] * numPounds
        print("That'll be %f please" % (cost))

# Основная функция
if __name__ == '__main__':
```

```
buyFruit('apples', 2.4)                # вызов функции buyFruit
buyFruit('coconuts', 2)
```

Проверка __name__ == '__main__' используется для разграничения выражений, которые выполняются, когда файл вызывается как сценарий из командной строки. Код после проверки – это код основной функции. Сохраните этот скрипт как fruit.py и запустите его:

```
C:\user\user\ai\lab1> python fruit.py
That'll be 4.800000 please
Sorry we don't have coconuts
```

### 1.2.7. Функционалы: filter и map. Лямбда выражения.

Ранее были представлены некоторые средства программирования в функциональном стиле — применение спискового включения для фильтрации и отображения. Здесь продемонстрируем применение встроенных функций filter и map для фильтрации и отображения, соответственно.

Иногда необходимо передать в функцию через ее формальный параметр имя другой функции. Такой параметр называют *функциональным*, а функцию, принимающую этот параметр — *функционалом*. Функция также может возвращать в виде результата другую функцию. Такие функции называют функциями *с функциональным значением*. Вызов функции с функциональным значением может быть аргументом функционала, а также использоваться вместо имени функции в вызове. Функционалы и функции с функциональным значением относятся к инструментарию функционального программирования и называются функциями *высших порядков*.

Первым аргументом filter должна быть некоторая функция-предикат, которая получает один аргумент и возвращает True, если значение должно включаться в результат. Например, такой функцией-предикатом может быть is_odd(x):

```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
def is_odd(x):
    """Возвращает True только для нечетных x."""
    return x % 2 != 0
```

Теперь воспользуемся встроенной функцией filter для получения нечетных значений списка numbers. Вызов list(filter(is_odd, numbers)) вернет список [3, 7, 1, 9, 5]. Здесь функция is_odd возвращает True, если ее аргумент является нечетным. Функция filter вызывает is_odd по одному разу для каждого значения numbers. Функция filter возвращает итератор, так что для получения результатов filter нужно будет выполнить их перебор. Это еще один пример отложенного вычисления. Функция list перебирает результаты и создает список, в котором они содержатся.

Для простых функций (типа is_odd), возвращающих только *значение одного выражения*, можно использовать *лямбда-выражение* для определения встроенной функции в том месте, где она нужна, — обычно при передаче другой функции в

качестве параметра, например: list(filter(lambda x: x % 2 != 0, numbers)). Лямбда-выражение является *анонимной функцией*, то есть функцией, не имеющей имени. В вызове filter(lambda x: x % 2 != 0, numbers) первым аргументом является лямбда-выражение

```
lambda x: x % 2 != 0
```

После lambda следует разделяемый запятыми список параметров, двоеточие (:) и выражение. В данном случае список параметров состоит из одного параметра с именем x. Лямбда-выражение *неявно* возвращает значение своего выражения. Таким образом, любая простая функция в форме

```
def имя_функции(список_параметров):
    return выражение
```

может быть выражена в более компактной форме посредством лямбда-выражения

```
lambda список_параметров: выражение
```

Воспользуемся встроенной функцией map с лямбда-выражением для возведения в квадрат каждого значения из numbers:

```
>>> numbers=[10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
>>> list(map(lambda x: x ** 2, numbers))
[100, 9, 49, 1, 81, 16, 4, 64, 25, 36]
```

Первым аргументом функции map является функция, которая получает одно значение и возвращает новое значение — в данном случае лямбда-выражение, которое возводит свой аргумент в квадрат. Вторым аргументом является итерируемый объект с отображаемыми значениями. Функция map использует отложенное вычисление, поэтому возвращаемый map итератор передается функции list. Это позволит перебрать и создать список отображенных значений.

Предшествующие операции filter и map можно объединить следующим образом:

```
>>>list(map(lambda x: x ** 2,  filter(lambda x: x % 2 != 0, numbers)))
[9, 49, 1, 81, 25]
```

### 1.2.8. Классы и объекты Python

Класс инкапсулирует данные и функции для взаимодействия с этими данными. Пример определения класса FruitShop (магазин фруктов, класс определен в файле shop.py):

```
class FruitShop:

    def __init__(self, name, fruitPrices):            # конструктор класса
        """
```

```
            name: Название магазина фруктов
            fruitPrices: Словарь с ключами в виде названий фруктов
            и ценами в виде значений, например:
            {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}
        """

        self.fruitPrices = fruitPrices
        self.name = name
        print('Welcome to %s fruit shop' % (name))

    def getCostPerPound(self, fruit):
        """
            Возвращает стоимость фрукта 'fruit''
            если он в словаре, иначе - None
            fruit:строка с названием фрукта
        """
        if fruit not in self.fruitPrices:
            return None
        return self.fruitPrices[fruit]

    def getPriceOfOrder(self, orderList):
        """
            Возвращает стоимость заказа orderList, включая только
            фрукты, которые есть в магазине.
            orderList: Список из кортежей (fruit, numPounds)
        """
        totalCost = 0.0
        for fruit, numPounds in orderList:
            costPerPound = self.getCostPerPound(fruit)
            if costPerPound != None:
                totalCost += numPounds * costPerPound
        return totalCost

    def getName(self):
        return self.name
```

Класс FruitShop содержит название магазина и цены за фунт некоторых фруктов, а также предоставляет функции или методы для этих данных. В чем преимущество обертывания этих данных в класс?

- инкапсуляция данных предотвращает их изменение или ненадлежащее использование;

- абстракция, которую предоставляют классы, упрощает написание кода общего назначения.

### Использование объектов

Как создать объект (экземпляр класса) и использовать его? Убедитесь, что у вас есть реализация FruitShop в shop.py. Затем импортируете код из этого файла (делая его доступным для других скриптов) с помощью import shop. Затем создайте объект типа FruitShop следующим образом:

```
import shop

# создание объекта-магазина 1 и его обработка
shopName = 'the Berkeley Bowl'
fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}
berkeleyShop = shop.FruitShop(shopName, fruitPrices)
applePrice = berkeleyShop.getCostPerPound('apples')
print(applePrice)
print('Apples cost $%.2f at %s.' % (applePrice, shopName))

# создание объекта- магазина  2 и его обработка
otherName = 'the Stanford Mall'
otherFruitPrices = {'kiwis': 6.00, 'apples': 4.50, 'peaches': 8.75}
otherFruitShop = shop.FruitShop(otherName, otherFruitPrices)
otherPrice = otherFruitShop.getCostPerPound('apples')
print(otherPrice)
print('Apples cost $%.2f at %s.' % (otherPrice, otherName))
print("That's expensive!")
```

Этот код находится в файле shopTest.py, запустить его можно так:

```
C:\user\user\ai\lab1>  python shopTest.py
Welcome to the Berkeley Bowl fruit shop
1.0
Apples cost $1.00 at the Berkeley Bowl.
Welcome to the Stanford Mall fruit shop
4.5
Apples cost $4.50 at the Stanford Mall.
That's expensive!
```

Проанализируем код. Оператор import shop обеспечивает загрузку определения класса из shop.py. Строка berkeleyShop = shop.FruitShop(shopName, fruitPrices) создает экземпляр (объект) класса FruitShop, определенного в shop.py, путем вызова конструктора __init__ этого класса. Обратите внимание, что мы передали только два аргумента, в то время как __init__ принимает три аргумента: (self, name, fruitPrices). Причина этого в том, что все методы в классе имеют в качестве первого аргумента self. Значение переменной self автоматически присваивается самому объекту; при вызове метода вы предоставляете только оставшиеся аргументы. Переменная self содержит все данные (name и fruitPrices) для текущего конкретного экземпляра.

### Статические переменные класса и переменные экземпляра класса

В следующем примере показано, как использовать статические переменные класса и переменные экземпляра класса в Python. Создайте файл person_class.py, содержащий следующий код:

```
class Person:
```

```
    population = 0

    def __init__(self, myAge):
        self.age = myAge
        Person.population += 1

    def get_population(self):
        return Person.population

    def get_age(self):
        return self.age
```

Выполним скрипт:

```
C:\user\user\ai\lab1> python person_class.py
```

Используем класс, следующим образом:

```
>>> import person_class
>>> p1 = person_class.Person(12)
>>> p1.get_population()
1
>>> p2 = person_class.Person(63)
>>> p1.get_population()
2
>>> p2.get_population()
2
>>> p1.get_age()
12
>>> p2.get_age()
63
```

В приведенном выше коде age (возраст) – это переменная экземпляра класса, а population (население) – статическая переменная класса. Статическая переменная population используется всеми экземплярами класса Person, тогда как каждый экземпляр имеет свою собственную переменную age.

### 1.2.9. Дополнительные советы и хитрости Python

Рассмотрим несколько полезных советов.

1. Применяйте функцию range для генерации последовательности целых чисел, используемых в качестве значений переменной цикла for. Последовательность справа от ключевого слова in в команде for должна быть *итерируемым объектом*, то есть объектом, из которого команда for может брать элементы по одному, пока не будет обработан последний элемент. Итератор напоминает закладку в книге — он всегда знает текущую позицию последовательности, чтобы вернуть следующий элемент по требованию.

Рассмотрим цикл for и встроенную *функцию* range для выполнения ровно 10 итераций с выводом значений от 0 до 9:

```
>>> for counter in range(10):
        print(counter, end=' ')

0 1 2 3 4 5 6 7 8 9
```

Вызов функции range(10) создает итерируемый объект, который представляет последовательность целых чисел 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Команда for прекращает работу при завершении обработки последнего целого числа, выданного range.

2. После импорта файла, если вы редактируете исходный файл, изменения не будут немедленно переданы в интерпретатор. Для этого используйте команду перезагрузки:

```
>>> reload(shop)
```

3. Рассмотрим некоторые проблемы, с которыми обычно сталкиваются изучающие Python.

**Проблема:** ImportError: нет модуля с именем py

**Решение**: для операторов импорта import <*имя-пакета*> не включайте в имя пакета расширение файла (т. е. подстроку .py).

**Проблема**: NameError: имя «MY_VAR» не определено. Даже после выполненного импорта вы можете увидеть такое сообщение.

**Решение**: чтобы получить доступ к элементу модуля, вы должны использовать module_name.member_name, где module_name - это имя файла, а member_name - имя переменной (или функции), к которой вы пытаетесь получить доступ.

**Проблема**: TypeError: объект dict не может быть вызван.

**Решение**: поиск элементов в словаре выполняется с использованием квадратных скобок [ ], а не круглых скобок () .

**Проблема:** ValueError: слишком много значений для распаковки.

**Решение**: убедитесь, что количество переменных, назначаемых в цикле for, совпадает с количеством элементов в каждом элементе списка. Аналогично при работе с кортежами. Например, если пара является кортежем из двух элементов (например, pair = ('apple', 2.0)), то следующий код вызовет ошибку “not enough values to unpack ”:

```
(a, b, c) = pair
```

Вот проблемный сценарий, связанный с циклом for:

```
pairList = [('apples', 2.00), ('oranges', 1.50), ('pears', 1.75)]
for fruit, price, color in pairList:
    print('%s fruit costs %f and is the color %s' % (fruit, price, color))
```

ValueError: not enough values to unpack (expected 3, got 2)

### 1.3. Задания для выполнения

#### Задание 1. Строки

Используя команды dir и help, изучите следующие методы строкового типа: ‘format’, ‘strip’, ‘lstrip’, ‘rstrip’, ‘capitalize’, ‘title’, ‘count’, ‘index’, ‘rindex’, ‘startswith’, ‘endswith’, ‘replace’, ‘split’, ‘rsplit’, ‘join’, ‘partition’, ‘rpartition’. Разработайте примеры вызова указанных методов и внесите результаты в отчет.

#### Задание 2. Списки

Используя команды dir и help, изучите, изучите следующие методы обработки списков: 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort'. Разработайте примеры вызова указанных методов и внесите результаты в отчет.

#### Задание 3. Словари

Используя команды dir и help, изучите, изучите следующие методы обработки словарей: 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', setdefault', 'update', 'values'. Разработайте примеры вызова указанных методов и внесите результаты в отчет.

#### Задание 4. Списковое включение

Определите списковое включение, которое из списка строк генерирует версию нового списка, состоящего из строк, длина которых больше пяти и которые записаны символами нижнего регистра. Решение можно проверить, просмотрев файл listcomp2.py.

#### Задание 5. Быстрая сортировка

Напишите функцию быстрой сортировки на Python, используя списки. Используйте первый элемент как точку деления списка. Вы можете проверить своё решение, просмотрев файл quickSort.py.

#### Задание 6. Решение задач и автооценивание

Чтобы ознакомиться с автооцениванием, мы попросим вас написать код, протестировать и включить в отчет решения **для трех задач, приведенных ниже**. Загрузите архив (codingdiagnostic_r.zip) с системой автооценивания в **отдельную рабочую папку**. Разархивируйте его и изучите содержимое. Архив содержит ряд файлов, которые вы будете *редактировать или запускать*:

addition.py: исходный файл для задачи (вопроса) 1
buyLotsOfFruit.py: исходный файл для задачи (вопроса) 2
shop.py: исходный файл для задачи (вопроса) 3
shopSmart.py: исходный файл для задачи (вопроса) 3
autograder.py: скрипт автооценивания (см. ниже)

Другие файлы в архиве можно *игнорировать*:

test_cases: каталог содержит тестовые примеры для каждой задачи (вопроса)
grading.py: код автооценивателя
testClasses.py: код автооценивателя
tutorialTestClasses.py: тестовые классы для этого лабораторного задания
projectParams.py: параметры

Команда python autograder.py оценит ваше решение для всех трех задач (вопросов). Если запустить автооцениватель перед редактированием каких-либо файлов, то получим ответ, фрагмент которого приведен ниже:

```
>>>  python autograder.py
Starting on 8-20 at 19:33:19

Question q1
===========

*** FAIL: test_cases\q1\addition1.test
***      add(a,b) must return the sum of a and b
***      student result: "0"
***      correct result: "2"
*** FAIL: test_cases\q1\addition2.test
***      add(a,b) must return the sum of a and b
***      student result: "0"
***      correct result: "5"
*** FAIL: test_cases\q1\addition3.test
***      add(a,b) must return the sum of a and b
***      student result: "0"
***      correct result: "7.9"
*** Tests failed.

### Question q1: 0/1 ###

Question q2
===========

*** FAIL: test_cases\q2\food_price1.test
,,,,

### Question q2: 0/1 ###

Question q3
===========

Welcome to shop1 fruit shop
…
***      correct result: "<FruitShop: shop3>"
```

```
*** Tests failed.

### Question q3: 0/1 ###

Finished at 19:33:19

Provisional grades
==================
Question q1: 0/1
Question q2: 0/1
Question q3: 0/1
------------------
Total: 0/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Для каждой из трех задач (вопросов) отображены результаты тестов, оценка и окончательное резюме в конце. Поскольку задачи еще не решены, все тесты не пройденны. По мере решения каждой задачи вы можете обнаружить, что некоторые тесты проходят успешно, а другие - нет. Когда все тесты будут пройдены, вы получаете итоговую оценку. Глядя на результаты для задачи (вопроса 1), вы можете увидеть, что не пройдены три теста с сообщением об ошибке «add (a, b) must return the sum of a and b». Ваш код всегда дает результат 0, но правильный ответ имеет другое значение. Это можно исправить следующим образом.

**Задача 1. Функция сложения**

Откройте addition.py и посмотрите шаблон определения функции add:

```
def add(a, b):
    "Возвращает сумму a и b"
    "*** ВСТАВЬТЕ ВАШ КОД СЮДА***"
    return 0
```

Тесты вызвали эту функцию с разными значениями a и b, но код всегда возвращал ноль. Измените это определение на следующее:

```
def add(a, b):
    ""Возвращает сумму a и b"
    print("Passed a = %s and b = %s, returning a + b = %s" % (a, b, a + b))
    return a + b
```

Теперь перезапустите автооцениватель и получите следующий результат (без результатов тестов для вопросов 2 и 3):

```
>>> python autograder.py -q q1
Starting on 8-20 at 19:47:54

Question q1
===========
```

```
Passed a = 1 and b = 1, returning a + b = 2
*** PASS: test_cases\q1\addition1.test
***      add(a,b) returns the sum of a and b
Passed a = 2 and b = 3, returning a + b = 5
*** PASS: test_cases\q1\addition2.test
***      add(a,b) returns the sum of a and b
Passed a = 10 and b = -2.1, returning a + b = 7.9
*** PASS: test_cases\q1\addition3.test
***      add(a,b) returns the sum of a and b

### Question q1: 1/1 ###
…
Finished at 19:47:54

Provisional grades
==================
Question q1: 1/1
Question q2: 0/1
Question q3: 0/1
------------------
Total: 1/3
```

Теперь вы прошли все тесты и получили итоговую оценку за решение задачи 1. Обратите внимание на новые строки «Passed a =…», которые появляются перед «*** PASS:…». Они создаются оператором печати в функции add. Вы можете использовать подобные операторы печати для вывода информации, полезной для отладки.

### Задача 2. Функция buyLotsOfFruit

Определите функцию buyLotsOfFruit(orderList) в файле buyLotsOfFruit.py, которая принимает список-заказ orderList, состоящий из кортежей (фрукт, вес), например,

```
orderList = [('apples', 2.0), ('pears', 3.0), ('lime', 4.0)]
```

и возвращает стоимость заказа. Для вычисления стоимости заказа функция использует ценник

```
fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75, 'limes': 0.75, 'strawberries': 1.00}
```

Если в списке-заказе есть название фрукта, которого нет в ценнике fruitPrices, то функция должна вывести сообщение об ошибке и вернуть None. Пожалуйста, не изменяйте переменную fruitPrices. Запускайте python autograder.py, пока не пройдут все тесты для задачи 2. Каждый тест подтверждает, что buyLotsOfFruit(orderList) возвращает правильный ответ с учетом различных возможных входных данных.

Например, test_cases / q2 / food_price1.test проверяет, равна ли стоимость списка заказа [('apples', 2.0), ('pears', 3.0), ('lime', 4.0)] значению 12,25

### Задача 3. Функция shopSmart

Определите функцию shopSmart(orderList, fruitShops) в файле shopSmart.py, которая принимает список заказов orderList и список магазинов FruitShops, и возвращает название магазина, где ваш заказ будет иметь наименьшую стоимость. Пожалуйста, не меняйте имя файла или имена переменных. Для решения задачи используйте класса FruitShop в файле shop.py (см. описание класса выше в п.1.2.8). Запускайте python autograder.py, пока не будут пройдены все тесты.

Каждый тест подтверждает, что shopSmart(orders, shops) возвращает правильный ответ с учетом различных возможных исходных данных. Например, при следующих значениях переменных

```
orders1= [('apples', 1.0), ('oranges', 3.0)]
orders2 = [('apples', 3.0)]
dir1 = {'apples': 2.0, 'oranges': 1.0}
shop1 = shop.FruitShop('shop1', dir1)
dir2 = {'apples': 1.0, 'oranges': 5.0}
shop2 = shop.FruitShop('shop2', dir2)
shops = [shop1, shop2]
```

Тест test_cases/q3/select_shop1.test проверяет, что shopSmart(orders1,shops) == shop1, а тест test_cases/q3/select_shop2.test проверяет, действительно ли shopSmart(orders2 ,shops) == shop2.

После завершения отладки скопируйте результаты автооценивания в отчет.

## 1.4. Порядок выполнения лабораторной работы

1.4.1. Изучите по материалам раздела 1.2 и [5] основы языка Python, структуры данных и методы обработки списков, кортежей, множеств, словарей, классы и объекты Python, среду программирования на языке Python. Проверьте выполнение приведенных примеров в среде программирования.

1.4.2. Выполните задания 1 – 5 из раздела 1.3 в интерактивном режиме, используя возможности IPython. Результаты выполнения каждого задания внесите в отчет.

1.4.3. Определите в соответствии с заданием 6 из раздела 1.3 функции и методы для решения 3-х сформулированных задач. Используйте для редактирования функций и выполнения кода интегрированную среду Spyder IDE. Зафиксируйте результаты выполнения функций во всех необходимых режимах. Выполните с помощью autograder.py автооценивание. При обнаружении ошибок отредактируйте код. Результаты автооценивания внесите в отчет.

## 1.5. Содержание отчета

Цель работы, задания 1-5 с примерами выполнения, описание класса для задания 6 и определений собственных функций, результаты выполнения всех собственных функций задания 6, результаты автооценивания задания 6, выводы.

### 1.6. Контрольные вопросы

1.6.1. Какие типы числовых данных поддерживает Python? Как определить тип переменной в Python?

1.6.2. Как выполнить целочисленное деление и вычисление остатка от деления целых чисел?

1.6.3. Приведите примеры использования логических операций и операций отношений.

1.6.4. Как выполнить конкатенацию строк, заменить символы нижнего регистра на верхний и наоборот, определить длину строки?

1.6.5. Приведите пример строки форматированного вывода в операторе print, объясните спецификации форматирования и назначение управляющих последовательностей.

1.6.6. Что такое f-стока? Приведите примеры использования.

1.6.7. Объясните назначение методов для работы со строками: 'format', ' 'index', 'isalpha', 'isdigit', 'partition', 'replace', 'split', 'title', 'translate', 'upper', 'zfill'.

1.6.8. Как создать список? Как выполняется обращение к элементам списков, сегментам списков? Объясните назначение методов для работы со списками: pop, append, insert, sort, index.

1.6.9. Как создать кортеж? Как его распаковать? Как обратиться к элементам кортежа? Как объединить список и кортеж?

1.6.10. Как создать множество? Как проверить принадлежность элемента множеству? Приведите примеры операций с множествами.

1.6.11. Что такое словарь? Как его создать? Как выполнить поиск в словаре? Как заменить значение в словаре? Как удалить элемент словаря? Как получить список ключей, список значений, список кортежей ключ-значение?

1.6.12. Напишите скрипт, печатающий номера элементов списка и сами элементы.

1.6.13.Что такое списковое включение? Приведите примеры операций отображения и фильтрации с помощью спискового включения.

1.6.14. Что такое выражение-генератор? Приведите пример применения в цикле for.

1.6.15. Как определить собственную функцию в Python? Определите функцию быстрой сортировки.

1.6.16. Что такое функционал? Как определяется анонимная функция? Приведите примеры использования функционалов filter и map.

1.6.17. Как определяется класс в языке Python? Приведите пример определения класса. Что такое статическая переменная класса? Чем она отличается от переменной экземпляра класса?