

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего
образования
«Северо-Кавказский федеральный университет»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению практических занятий
по дисциплине «Алгоритмизация и программирование»
для студентов направления
13.03.02 «Электроэнергетика и электротехника»
Направленность (профиль): «Цифровые технологии в
электроэнергетике»

Ставрополь 2025

СОДЕРЖАНИЕ

Практическое занятие №1	4
Теоретическая часть	4
Примеры решения задания	15
Задание для самостоятельного выполнения	17
Контрольные вопросы.....	18
Практическое занятие №2	20
Теоретическая часть	20
Примеры решения задания	32
Задание для самостоятельного выполнения	35
Задание.....	35
Контрольные вопросы.....	37
Практическое занятие №3	38
Теоретическая часть	38
Примеры решения задания	41
Задание для самостоятельного выполнения	41
Контрольные вопросы.....	44
Практическое занятие №4	45
Теоретическая часть	45
Примеры решения задания	50
Задание для самостоятельного выполнения	50
Контрольные вопросы.....	52
Практическое занятие №5	53
Теоретическая часть	53
Примеры решения задания	76
Задание для самостоятельного выполнения	77
Контрольные вопросы.....	79
Практическое занятие №6	80
Теоретическая часть	80
Примеры решения задания	90

Задание для самостоятельного выполнения	90
Контрольные вопросы.....	99
Практическое занятие №7	100
Теоретическая часть	100
Примеры решения задания	102
Задание для самостоятельного выполнения	105
Контрольные вопросы.....	105
Практическое занятие №8	108
Теоретическая часть	108
Примеры решения задания	111
Задание для самостоятельного выполнения	111
Контрольные вопросы.....	111
Практическое занятие №9	116
Теоретическая часть	116
Примеры решения задания	117
Задание для самостоятельного выполнения	118
Контрольные вопросы.....	118
СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ	122

Практическое занятие №1

Тема: Знакомство со средой программирования Python IDLE и облачными онлайн интерпретаторами языка Python

Цель работы: Получить основные сведения о среде программирования Python IDLE, познакомиться с возможностями интегрированной среды, упрощающими разработку программы, рассмотреть простейшие программы ввода-вывода данных.

Теоретическая часть

Для создания программы на любом языке программирования необходимы всего лишь две программы:

1. Текстовый редактор, например, «Блокнот», в котором будет писаться код программы.

2. Программа – «переводчик» (компилятор или интерпретатор) написанных текстовых файлов с языка программирования на машинный язык.

Тем не менее, блокнотом обычно никто не пользуется, а пользуются специализированными редакторами, которые имеют ряд дополнительных удобных функций:

- подсвечивание ошибок;
- подсвечивание ключевых слов;
- авто заполнение по начальным символам фразы;
- прочее.

Помимо специализированных редакторов разработчики часто пользуются дополнительными утилитами, которые позволяют упрощать и ускорять процесс разработки. К ним относятся:

- программы–отладчики, которые позволяют останавливать код программы в нужном месте и смотреть значения переменных в этот момент;
- редакторы ресурсов;

- прочие дополнительные утилиты.

Объединение всех этих утилит обычно называется интегрированная среда разработки, или IDE (Integrated Development Environment).

Для языка Python существует множество платных и бесплатных интегрированных сред разработки. Например:

- PyCharm
- PyDev
- WingWare
- Komodo IDE
- VS Code
- Spyder
- IDLE
- и другие.

Наиболее простой и удобной для начинающего программиста является IDE IDLE. IDLE (Integrated Development and Learning Environment) – это интегрированная среда для разработки (и обучения), которая поставляется вместе с Python. Сама IDLE написана на Python с использованием библиотеки Tkinter, поэтому является кроссплатформенным приложением (может работать на Windows, Mac OS, Linux).

IDLE – это, по сути, текстовый редактор с подсветкой синтаксиса, автозаполнением, умным отступом и другими функциями для удобства написания кода на Python.

Скачать самую последнюю версию интерпретатора языка Python вместе с интегрированной средой программирования IDLE можно совершенно бесплатно на официальном сайте проекта: <https://www.python.org>. При установке рекомендуется установить галочки «Install launcher for all users» и «Add Python to PATH» (рисунок 1).



Рисунок 1 – Установка интерпретатора языка Python

IDLE в Windows находится в меню «Пуск» → «Python 3.x» → «IDLE». Также можно быстро найти его через «Поиск» около меню «Пуск», набрав в поле поиска «IDLE»:

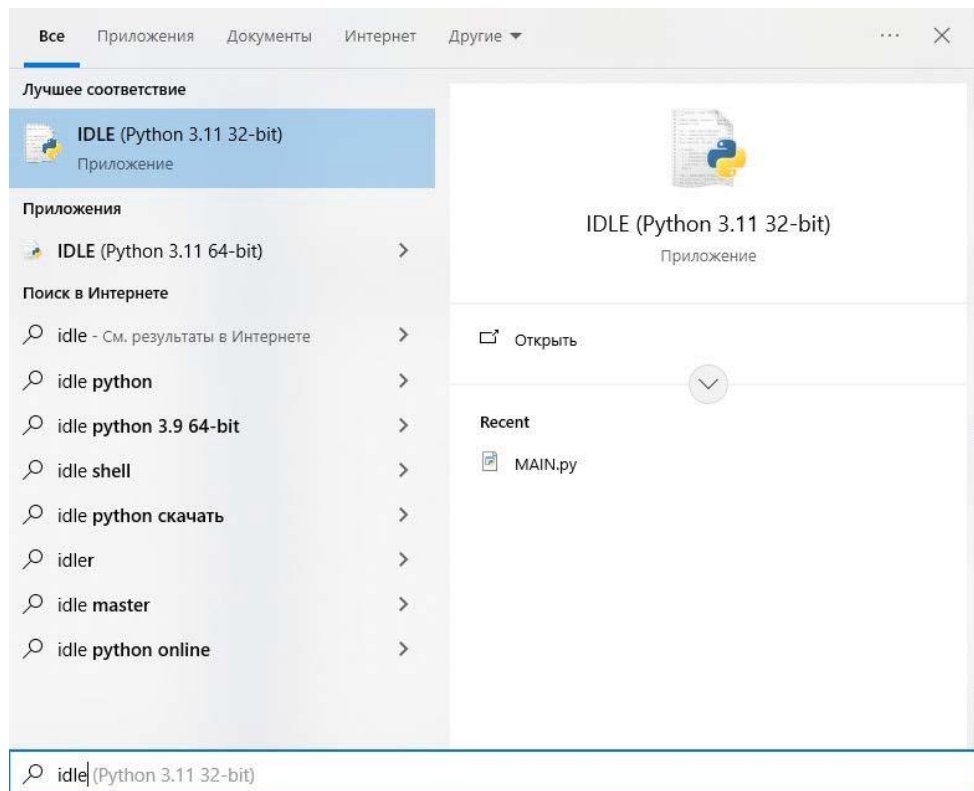


Рисунок 2 – Запуск IDLE через меню поиска

Откроется интегрированная среда для разработки и обучения IDLE, изображенная на рисунке 3.

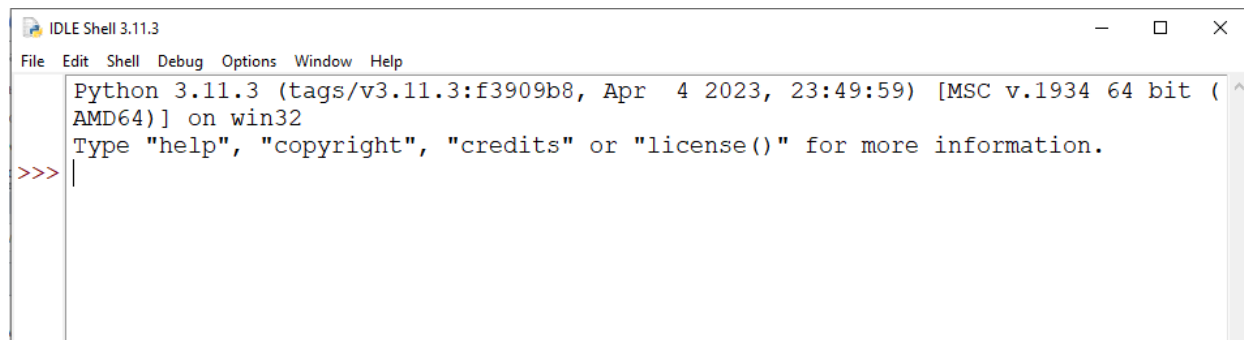


Рисунок 3 – Интегрированная среда разработки IDLE

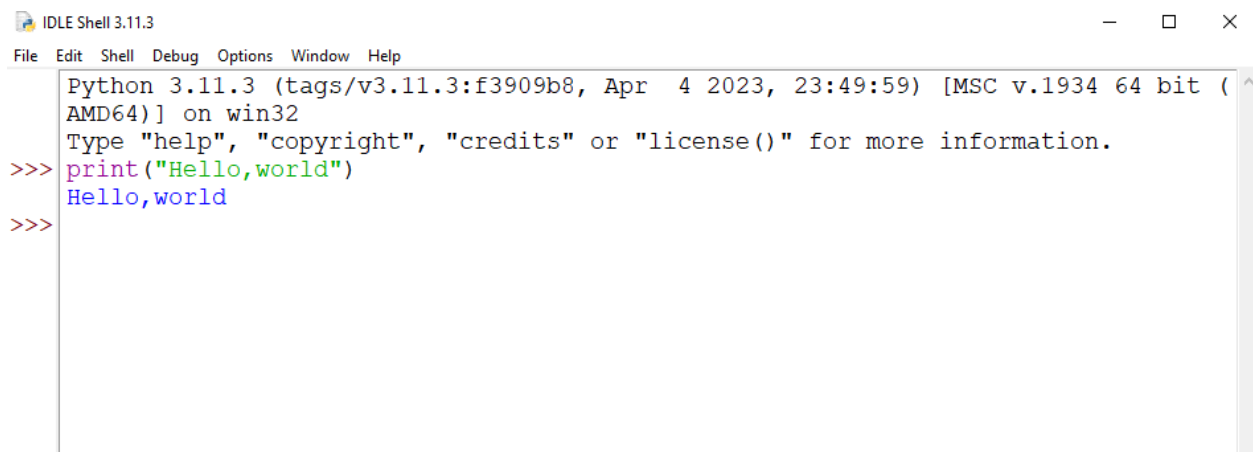
IDLE поддерживает 2 режима работы:

1. Интерактивный режим. В этом режиме команды выполняются сразу после их вызова. Результат выполнения команд сразу выводится на экран. Поскольку в названии среды есть слово «Learning» (обучение), этот режим еще можно назвать режимом обучения. Он хорошо подходит для написания простых программ и тестирования работы фрагментов кода. Интерактивный режим имеет следующие взаимосвязанные недостатки:

- программный код, который вводится, нигде не сохраняется. Чтобы повторно запустить один и тот же программный код, его нужно снова ввести;
- программы, которые введены в интерактивном режиме, после выполнения интерпретатором Python исчезают. Чтобы выполнить их повторно, нужно их снова набрать или использовать операции копирования, что неэффективно, либо использовать сочетания клавишу «Alt»+«n», «Alt»+«p» для перебора вперед и назад ранее введенных команд, что тоже не всегда удобно.

2. Файловый режим. В этом режиме сначала записывается вся программа, а потом эта программа выполняется полностью. Предварительно текст программы нужно сохранить. Программа на языке Python носит имя «сценарий» или «скрипт».

Изначально программа запускается в интерактивном режиме. Все что вы напишете в интерактивном режиме после ">>>" будет сразу же выполнено построчно. Традиционный "**Hello world!**" будет выглядеть так:



The screenshot shows the IDLE Shell 3.11.3 window. The title bar reads "IDLE Shell 3.11.3". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area displays the following text: "Python 3.11.3 (tags/v3.11.3:f3909b8, Apr 4 2023, 23:49:59) [MSC v.1934 64 bit (AMD64)] on win32", "Type 'help', 'copyright', 'credits' or 'license()' for more information.", and a prompt ">>>". Below the prompt, the code `print("Hello,world")` is entered, and the output "Hello,world" is displayed. Another prompt ">>>" is visible on the next line.

Рисунок 4 – Пример простейшей программы

Файловый режим можно вызвать с помощью команды «File→New File». Откроется текстовый редактор, изображенный на рисунке 5.

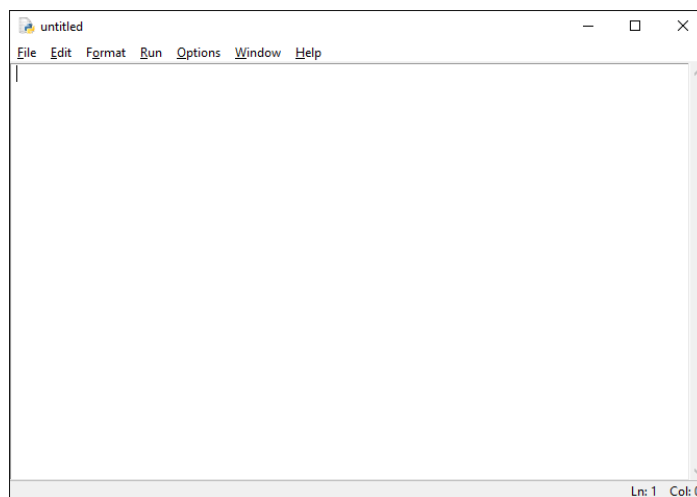


Рисунок 5 – Редактор исходного кода программы IDLE

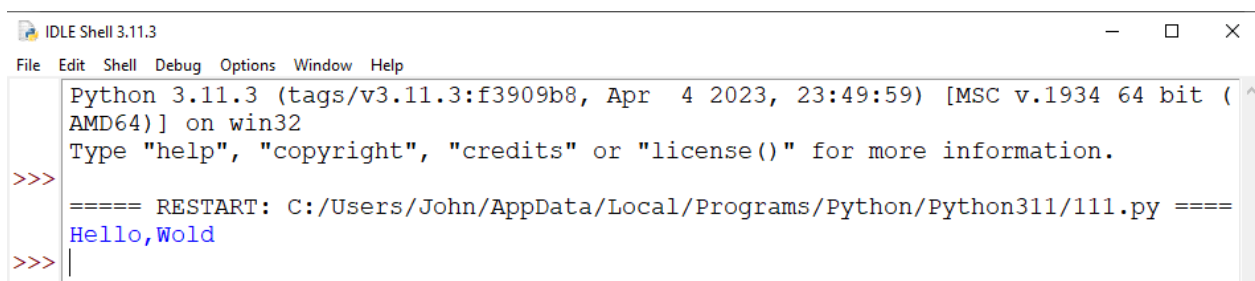
Введем в открывшееся окно текст программы. Например, введенную ранее команду `print("Hello,World")` (рисунок 6). Эту команду можно либо ввести вручную, либо скопировать из предыдущего окна.



Рисунок 6 – Создание программы в файловом режиме

Примечание: для использования горячих клавиш копировать/вставить Ctrl+C, Ctrl+V обязательно должна быть выбрана английская раскладка клавиатуры. В случае, если выбрана русская раскладка клавиатуры горячие сочетания клавиш работать не будут. В этом случае можно использовать команды контекстного меню «Copy»/«Paste».

После того, как текст программы будет введен, необходимо выполнить команду меню «Run→Run Module» либо нажать клавишу «F5». Программа попросит выбрать путь для сохранения текста скрипта, после чего результаты выполнения программы будут отображены в окне среды IDLE (рисунок 7).



```
Python 3.11.3 (tags/v3.11.3:f3909b8, Apr 4 2023, 23:49:59) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:/Users/John/AppData/Local/Programs/Python/Python311/111.py ====
Hello,World
>>> |
```

Рисунок 7 – Результат выполнения программы в файловом режиме

Независимо от режима работы, интегрированная среда IDLE имеет ряд функций, которая упрощает разработку программы:

1. Подсветка синтаксиса.

Чтение и написание кода становится более удобным – каждая конструкция языка подсвечивается определенным цветом. Комментарии красным, строки зеленым, вывод синим и т.д (рисунок 8).

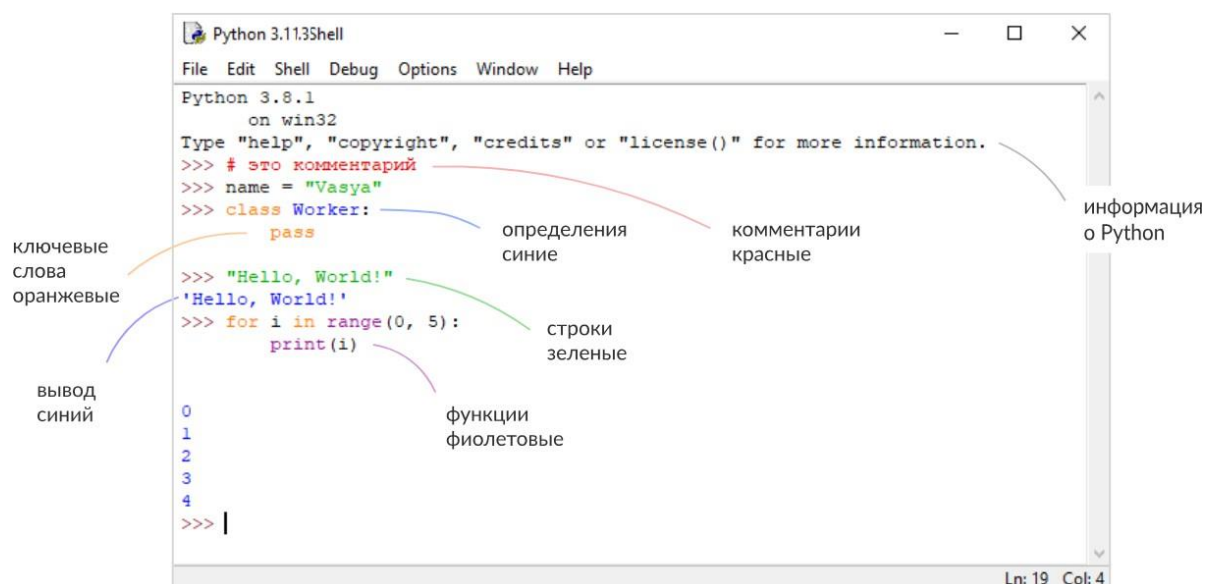


Рисунок 8 – Подсветка синтаксиса в IDLE

2. Автоматические отступы.

Как вы знаете, в языке Python отступы в тексте с начала строки являются частью синтаксиса языка и играют ту же роль, что фигурные скобки в языках типа «Си» или ключевые слова begin/end в языках типа «Pascal». В связи с этим, если строка заканчивается символом «:», при переходе на следующую строку будет автоматически добавлен отступ (рисунок 9). По умолчанию их размер равен 8-ми пробелам.

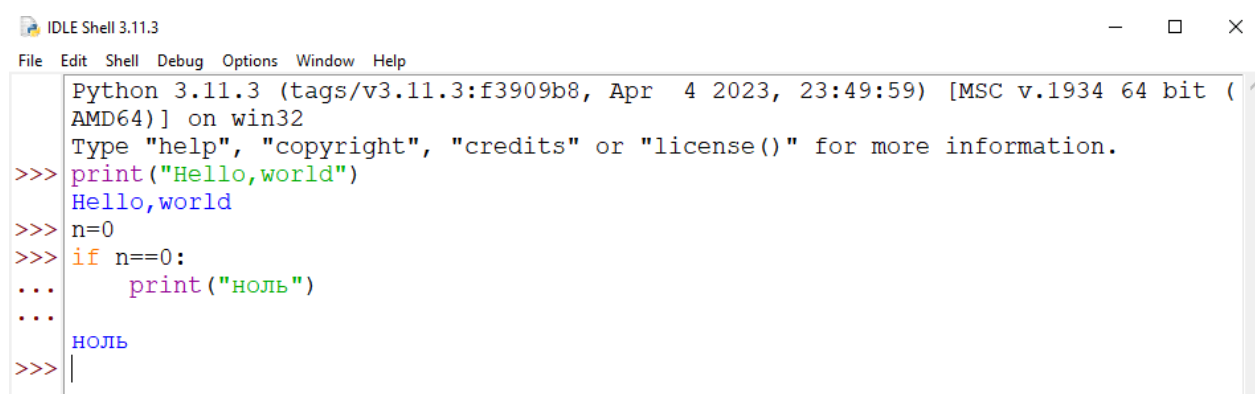


Рисунок 9 – IDLE автоматически создает отступы для многострочных инструкций

При нажатии на «enter» в нужных конструкциях, отступы создаются автоматически. Если вам необходимо сдвинуться на предыдущий уровень,

нажмите «Backspace». Если необходимо сдвинуться вправо, нажмите «Tab». Если вы напечатали выражение «pass», «return», «break», «continue» или «raise» и нажали «enter», курсор автоматически вернется к прежнему отступу.

3. Подсказки к функциям.

При вызове функции или метода, после открывающейся скобки (где далее будут прописаны аргументы) IDLE отобразит подсказку. В ней будут описаны аргументы, которые ожидает функция (рисунок 10). Подсказка будет отображаться до тех пор, пока вы не закроете скобку.

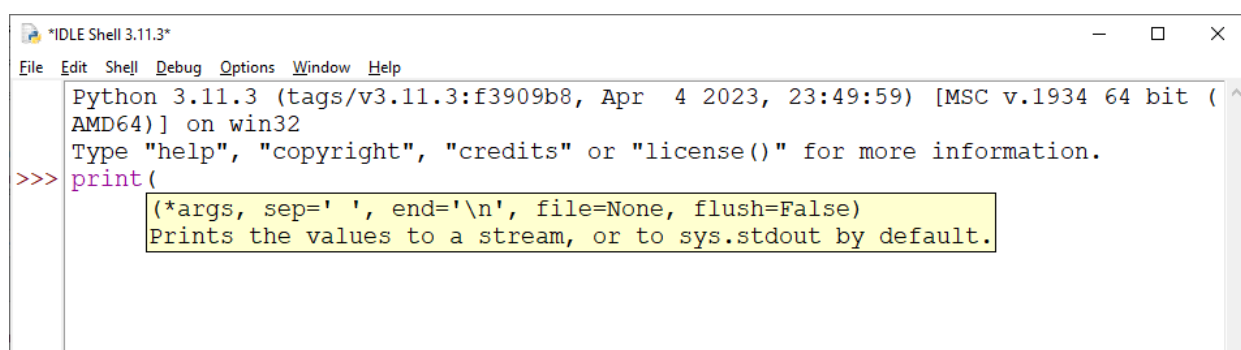


Рисунок 10 – Подсказки для функций в IDLE Python

4. Отступ для нескольких команд кода

Иногда, в процессе разработки программы, возникает ситуация, когда необходимо сдвинуть вправо или влево сразу несколько строчек кода. Например, в случае, если необходимо добавить условие для выполнения определенного участка кода, состоящего из нескольких строк. Для этого необходимо добавить оператор `if`, и каждую строчку, относящийся к нему сдвинуть вправо с помощью клавиши `Tab`. Однако это не всегда бывает удобно, так как код может быть весьма объемным и занимать несколько десятков строк. Для упрощения этой операции, в среде разработки IDLE предусмотрена специальная команда «Format» → «Indent Region». Для ее использования необходимо выделить соответствующие строки, которые нуждаются в сдвиге, и выполнить команду «Format» → «Indent Region» из

меню окна редактора кода, либо воспользоваться горячим сочетанием клавиш «Ctrl»+ «]» (рисунок 11).

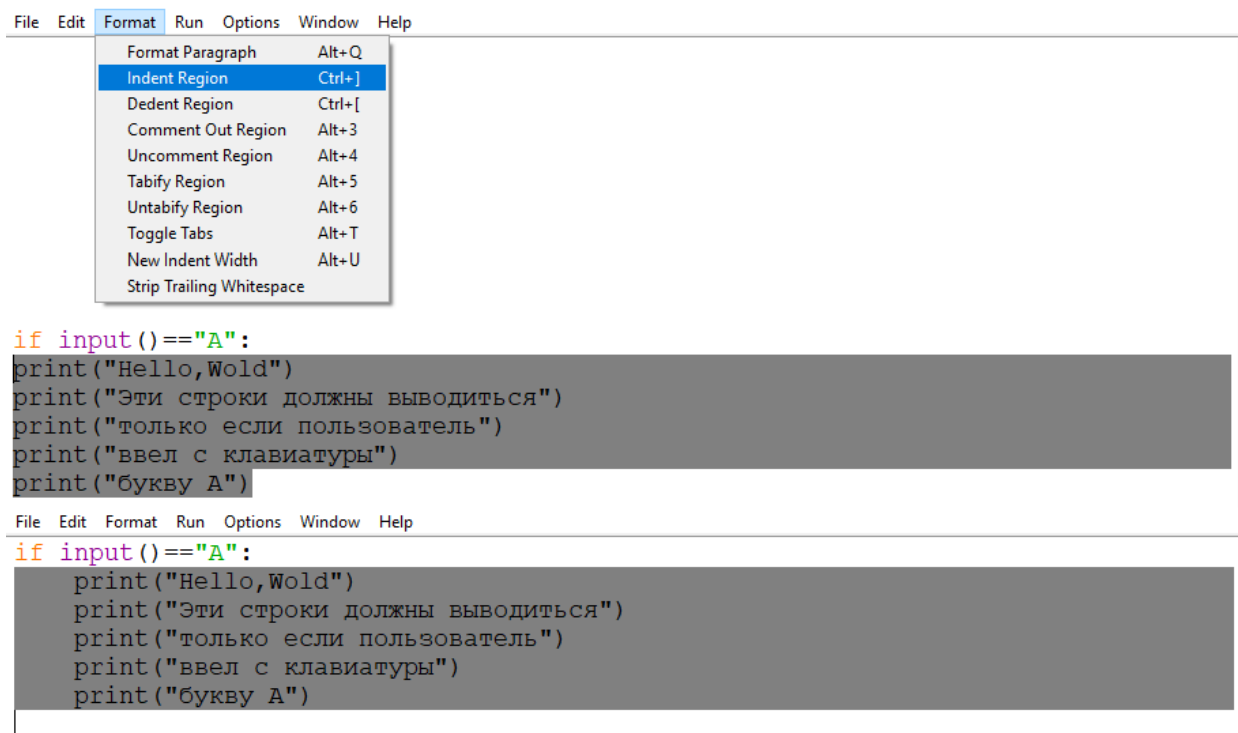


Рисунок 11 – Для сдвига вправо выделить строки и нажать «Ctrl»+ «]»

В тех случаях, когда необходимо несколько строк кода сдвинуть влево, существует обратная команда «Format» → «Dedent Region», либо сочетание горячих клавиш «Ctrl»+ «[».

5. Отключение блока кода.

Иногда в процессе разработки программы возникает необходимость временно отключить выполнение какого-то участка кода. Обычно это достигается путем «закомментирования» кода. В начале каждой строчки, которую необходимо на время отключить от выполнения пишется символ «#», что превращает эту строчку в комментарий и соответствующие команды не выполняются. Однако если нужно отключить выполнение большого участка кода такой способ может быть весьма трудоемким. Для упрощения этой операции в среде разработки IDLE есть специальные команды «Format» → «Comment Out Region» (закомментировать выделенную область) (горячие

клавиши Alt+3) и «Format» → «Uncomment Region» (раскомментировать) (горячие клавиши Alt+4) (рисунок 12).

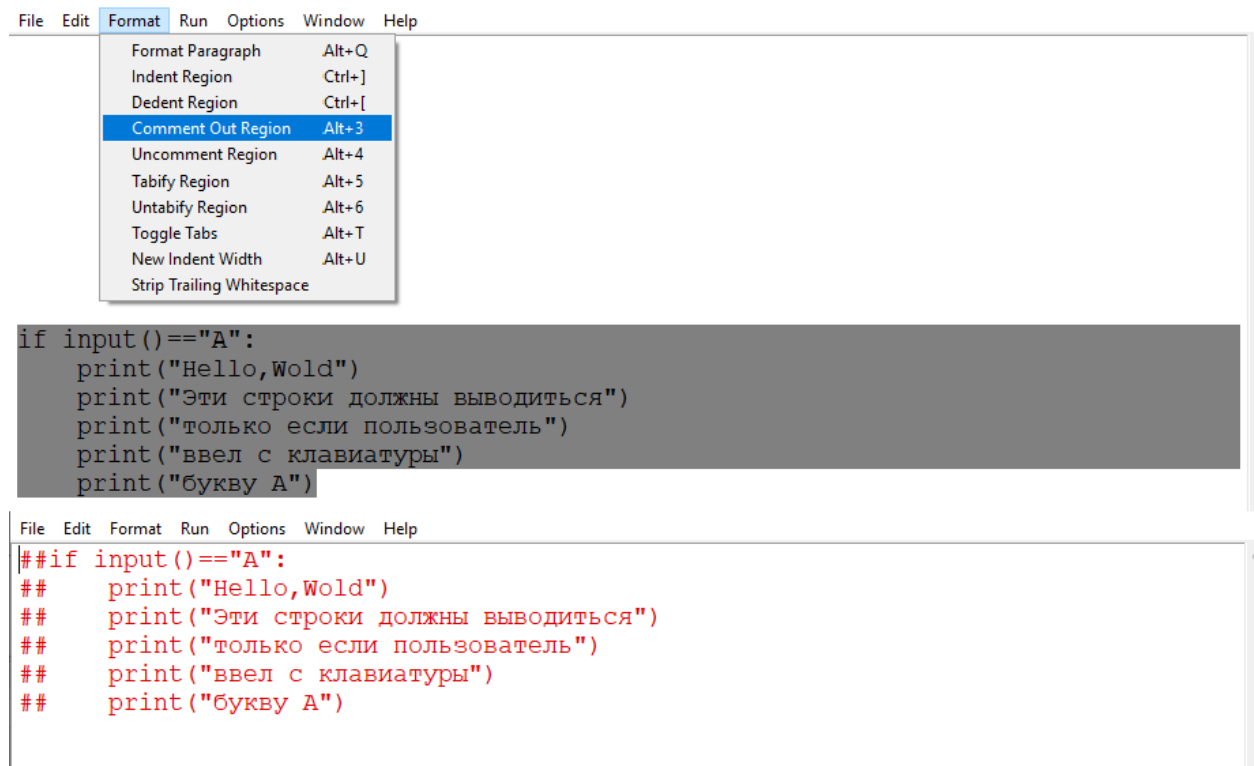


Рисунок 12 – Команда «закомментировать выделенную область»

6. Дополнительные настройки отображения.

Среда разработки IDLE позволяет также настроить дополнительные параметры отображения: изменить размер и стиль шрифта, цвета подсветки, сочетания горячих клавиш и так далее. Все эти настройки можно изменить через команду меню «Options» → «Configure IDLE» (рисунок 13).

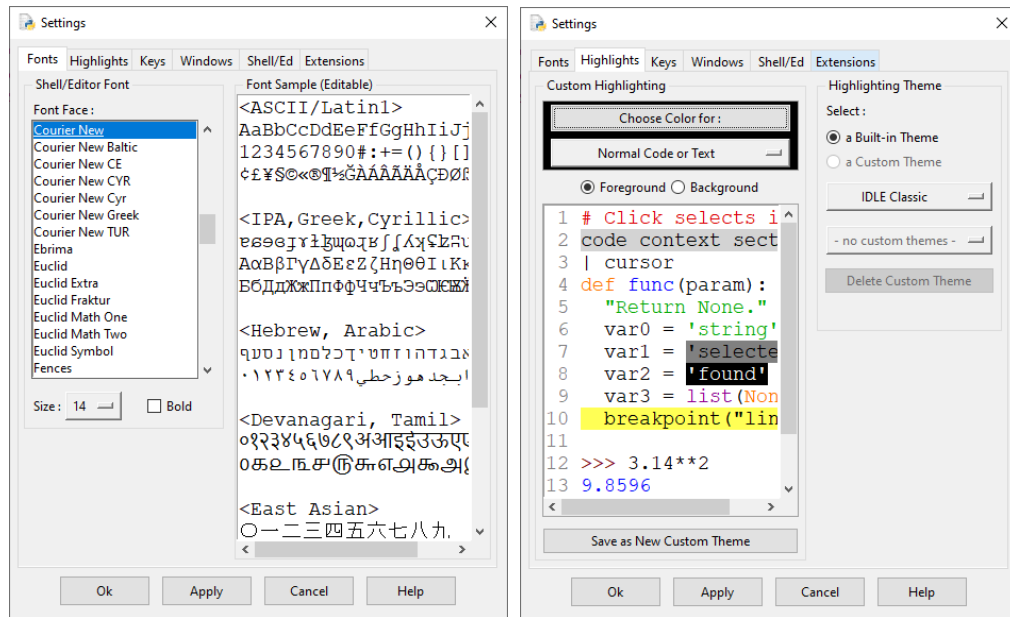


Рисунок 13 – Настройки среды разработки IDLE

Например, команда «Options» → «Show Line Numbers» позволяет добавить номера строк кода в редактор кода, чтобы бывает удобно при отладке и поиске ошибок в коде.

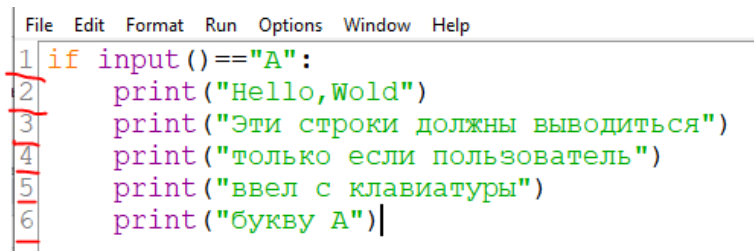


Рисунок 14 – Добавление нумерации строк кода

Онлайн среды разработки

Бывают ситуации, когда нет возможности установить на своем компьютере полноценную среду разработки и интерпретатор языка Python. В этом случае могут помочь многочисленные онлайн среды разработки, имеющиеся в сети интернет.

Например:

- <https://www.online-python.com/>
- https://www.onlinegdb.com/online_python_compiler
- <https://replit.com/languages/python3>

- <https://onlinepythoncompiler.com/>
- <https://www.codabrainy.com/python-compiler/>

Перечисленные ресурсы отличаются удобством интерфейса, скоростью работы, набором дополнительных функций и установленных компонентов языка Python.

Примеры решения задания

Задача: пользователь вводит с клавиатуры 5 чисел. Необходимо посчитать количество четных и нечетных чисел и вывести на экран это количество.

Решение:

Запустите интегрированную среду разработки IDLE и переключите ее в файловый режим с помощью команды «File→New File».

Для ввода с клавиатуры будет использовать функцию `input`.

Для преобразования введенной последовательности текста в целое число используем функцию `int`.

Для проверки числа на четность/нечетность будем использовать остаток от деления на 2.

Для подсчета количества четных и нечетных чисел введем две дополнительные переменные, которые будут увеличиваться на единицу, в зависимости от результатов проверки на четность или нечетность.

С учетом вышесказанного, программа будет выглядеть примерно следующим образом:

```

File Edit Format Run Options Window Help
1 a1=int(input("Введите число №1: "))
2 a2=int(input("Введите число №2: "))
3 a3=int(input("Введите число №3: "))
4 a4=int(input("Введите число №4: "))
5 a5=int(input("Введите число №5: "))
6 chet=0
7 nechet=0
8 if a1%2==0:
9     chet+=1
10 else:
11     nechet+=1
12 if a2%2==0:
13     chet+=1
14 else:
15     nechet+=1
16 if a3%2==0:
17     chet+=1
18 else:
19     nechet+=1
20 if a4%2==0:
21     chet+=1
22 else:
23     nechet+=1
24 if a5%2==0:
25     chet+=1
26 else:
27     nechet+=1
28 print("Четных чисел:", chet)
29 print("Нечетных чисел:", nechet)

```

Рисунок 15 – Пример решения задания

Введите этот код в редактор IDLE. Выполнить команду меню «Run→Run Module». Сохраните файл с кодом программы. После сохранения,

```

File Edit Shell Debug Options Window Help
Python 3.11.3 (tags/v3.11.3:f3909b8, Apr 4 2023, 23:49:59) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:/Users/John/AppData/Local/Programs/Python/Python311/chet.py ====
Введите число №1: 1
Введите число №2: 2
Введите число №3: 3
Введите число №4: 4
Введите число №5: 5
Четных чисел: 2
Нечетных чисел: 3
>>>

```

Рисунок 16 – Результат выполнения программы

Задание для самостоятельного выполнения

1. Ознакомьте с теоретическим материалом лекции №1.
2. Выполните индивидуальное задание, согласно вашему номеру варианта (Таблица 1).

Таблица 1 – Задания для самостоятельного выполнения:

№ варианта	Задачи для выполнения	№ варианта	Задачи для выполнения
1	9,6,10,11,12	9	8,9,2,11,12
2	10,7,4,11,12	10	5,6,7,11,12
3	7,5,2,11,12	11	9,5,6,11,12
4	6,8,9,11,12	12	5,7,2,11,12
5	6,7,2,11,12	13	7,5,1,11,12
6	7,10,1,11,12	14	3,9,4,11,12
7	9,10,1,11,12	15	6,7,8,11,12
8	10,8,2,11,12	16	1,2,3,11,12

Задача №1. Создайте программу решения квадратного уравнения. Коэффициенты А,В,С квадратного уравнения пользователь должен вводить с клавиатуры, корни уравнения необходимо вывести на экран.

Задача №2. Напишите следующую программу: пользователь вводит с клавиатуры 5 чисел. Необходимо найти их сумму и вывести на экран.

Задача №3. Напишите следующую программу: пользователь вводит с клавиатуры 5 чисел. Необходимо найти их произведений и вывести на экран.

Задача №4. Напишите следующую программу: пользователь вводит с клавиатуры 5 чисел. Необходимо посчитать, сколько чисел оказалось больше 10 и вывести на экран это количество.

Задача №5. Напишите следующую программу: пользователь вводит с клавиатуры 5 чисел. Необходимо посчитать количество четных и нечетных чисел и вывести на экран это количество.

Задача №6. Напишите следующую программу: пользователь вводит с клавиатуры 5 чисел. Необходимо вывести на экран сообщение, показывающее, каких чисел оказалось больше: четных или нечетных.

Задача №7. Напишите следующую программу: пользователь вводит с клавиатуры 5 чисел. Необходимо посчитать произведение четных и сумму нечетных чисел и вывести на экран полученные значения.

Задача №8. Напишите следующую программу: пользователь вводит с клавиатуры 5 чисел. Необходимо найти сумму квадратов этих чисел и вывести на экран.

Задача №9. Напишите следующую программу: пользователь вводит с клавиатуры длину и ширину прямоугольника. Необходимо посчитать и вывести на экран: периметр прямоугольника, его площадь и длину диагонали.

Задача №10. Напишите следующую программу: пользователь вводит с клавиатуры 5 чисел. Необходимо четные числа сложить по правилам сложения чисел, а нечетные числа сложить по правилам сложения строк.

Задача №11. Напишите программу, которая считывает целое число, после чего на экран выводится следующее четное и предыдущее нечетное целое число и соответствующий пояснительный текст.

Задача №12. Напишите программу, которая будет пересчитывать временной интервал, заданный в минутах, в величину, выраженную в часах и минутах. (Например 80 минут – это 1 час, 20 минут).

Контрольные вопросы

1. Что такое IDE? Какие IDE для программирования на Python вы знаете?
2. Какие режимы работы IDLE вы знаете? Чем они отличаются?
3. Какие возможности интегрированной среды разработки IDLE, упрощающие разработку программы, вам известны?

4. Как в среде разработки IDLE сделать отступ сразу для нескольких строк кода?

5. Как в среде разработки IDLE закомментировать/раскомментировать несколько строк кода?

Практическое занятие №2

Тема: Списки и циклы. Создание списков. Операции с элементами списков. Итерационный обход элементов списков

Цель работы: Получить основные о списках, кортежах, словарях, способах создания списков, методами работы со списками.

Теоретическая часть

В языке Python есть поддержка более сложных типов данных. Если у вас есть несколько предметов (например, несколько марок авто), то хранение автомобилей в отдельных переменных может выглядеть так:

```
car1 = "Ford"  
car2 = "Volvo"  
car3 = "BMW"
```

Для каждой новой марки нужно выделять переменную с новым именем. Это очень неудобно, ведь если их будет не 3, а, например, 3000, нужно будет создавать 3000 новых переменных. Кроме того, работать с такими переменными тоже будет неудобно. Например, если нужно будет проверить, есть ли среди перечисленных марок какая-то определенная, нужно будет написать 3000 команд сравнения. К счастью, существуют более простые подходы к написанию программы. Для таких случаев существует специальный тип переменных, который называется «список».

Список – это специальная переменная, которая может содержать более чем одно значение. Для создания списка необходимо использовать квадратные скобки. Например:

```
a=[]
```

Либо использовать ключевое слово `list`:

```
a=list()
```

Эти команды создадут пустой список. При создании списка можно сразу же поместить в него значения, через запятую. Например:

```
a=[1,2,3,10]
```

Вывести на экран все элементы списка можно с помощью уже известной нам команды `print`:

```
a=[1,2,3,10]  
print(a)
```

Обращение к элементам списка

Каждый элемент в списке имеет свой индекс. Индекс – это порядковый номер элемента в списке. Индекс начинается с нуля.

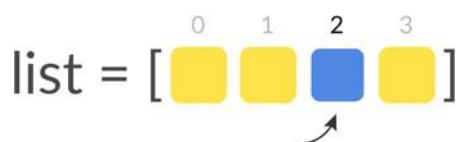


Рисунок 1 – Индексация элементов в списке

Для доступа к элементу списка по индексу используются квадратные скобки:

```
a=[1,2,3,10]  
print(a[2])
```

Указанный фрагмент кода выведет на экран число 3, так по индексу 2 располагается именно это число.

Существует также отрицательный индекс. Отрицательными индексами называют расположение элементов в списке справа налево. При этом нумерация начинается с -1.



Рисунок 2 – Индексация элементов в списке с конца

Отрицательным индексом удобно пользоваться, когда необходимо обратиться к последнему в списке элементу, не высчитывая его номер. Любой конечный элемент будет с индексом, равным -1.

Если же нужно получить из списка не один, а сразу несколько элементов, используются так называемые срезы. Срезы (slices) — это подмножества элементов списка. У них есть свой собственный синтаксис. Записывается срез так же, как обращение к элементу, используя индекс. **Пример:**

```
elements[START:STOP:STEP]
```

В этом случае берётся срез от номера start (включительно) до stop (не включая его), а step — это шаг. По умолчанию start и stop равны 0, step равен 1.

```
elements = [0.1, 0.2, 1, 2, 3, 4, 0.3, 0.4]
int_elements=elements[2:6]#выборка со 2 по 6-й элементы
print(int_elements)
```

На экран будут выведены элементы с индексами 2,3,4 и 5, то есть, в данном случае, только целые числа.

Добавление элементов в список

В списках доступно добавление, изменение, удаление элементов. Рассмотрим каждый способ изменения элементов на примерах.

Для того чтобы добавить новый элемент в список, используется `list.append(x)`, где `list` — список, `x` — нужное значение.

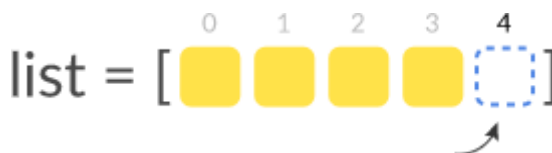


Рисунок 3 – Добавление элемента в конец списка

```
>>> elements = [1, 2, 3, 'word']
>>> elements.append('meow')
>>> print(elements)
```

```
[1, 2, 3, 'word', 'meow']
```

Для простого примера, рассмотрим создание списка с нуля с помощью метода **append**:

```
>>> elements = []
>>> elements.append(1)
>>> elements.append('word')
>>> elements.append('meow')
>>> print(elements)
[1, 'word', 'meow']
```

Функция **append** добавляет элемент в конец списка. Но если нужно добавить элемент в середину списка, необходимо использовать метод **insert**. Метод **insert** имеет следующий синтаксис:

```
list.insert(i, x)
```

где `list` – список,

`i` – позиция,

`x` – нужное значение.

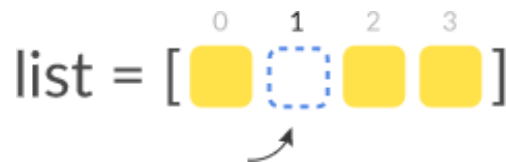


Рисунок 4 – Добавление элемента в середину списка

Пример:

```
>>> elements = [1, 2, 4]
>>> print(elements)
[1, 2, 4]

>>> elements.insert(2, 3)
>>> print(elements)
[1, 2, 3, 4]
```

Изменение элементов списка

Изменение элементов списка происходит следующим образом: нужно выбрать элемент по индексу (порядковому номеру элемента) и присвоить новое значение.



Рисунок 5 – Замена элемента в списке

Пример:

```
>>> elements = [2, 4, 6]
>>> elements[2] = 8

>>> print(elements)
[2, 4, 8]
```

В примере выше мы заменили 6 на 8. Не забывайте, что счёт начинается с нуля, и в данном списке цифра 6 это 2-й элемент.

Для удаление из списка используют функция **pop**, которая имеет следующий синтаксис:

```
list.pop(i)
```

где `list` – список,

`i` – индекс (позиция) элемента в списке.

Если не указывать индекс, то будет удаляться последний элемент из списка.



Рисунок 6 – Удаление последнего элемента списка

Объединение списков

Списки в Python можно объединять с помощью оператора + или метода extend .



Рисунок 7 – Объединение двух списков

Выглядит это так:

```
>>> a = [1, 3, 5]
>>> b = [1, 2, 4, 6]
>>> print(a + b)
[1, 3, 5, 1, 2, 4, 6]

>>> hello = ["h", "e", "l", "l", "o"]
>>> world = ["w", "o", "r", "l", "d"]
>>> hello.extend(world)      # extends не возвращает новый
список, а дополняет текущий
>>> print(hello)
['h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']
```

У списков есть еще несколько полезных функций:

list.clear() – удаляет все элементы списка, не имеет аргументов;

list.count(x) – посчитает количество элементов x в списке;

len(list) – посчитает количество всех элементов в списке;

list.remove(x) – удалить из элемент x (удаляет только один элемент, который встретится первый раз);

list.index(x) – вернет позицию первого найденного элемента x в списке;

list.copy() – служит для копирования списков;

list.reverse() – меняет порядок элементов в списке на противоположный;

list.sort() – сортирует список.

Полный список функций можно посмотреть с помощью команды help:

```
>>> help(list)
```

Копирование списков

Попробуем выполнить следующую программу:

```
a=1
b=a
b=0
print(a)
print(b)
```

На экране будет выведено два числа: 1 и 0. Теперь выполним следующую программу:

```
a=[1,2,3]
b=a
b[0]=0
print(a)
print(b)
```

В результате на экране появится две строчки: [0,2,3] и [0,2,3]. Такой результат возникает из-за особенностей языка Python. Оператор присвоения «=» для простейших и расширенных типов данных ведет себя по-разному: для простейших типов он перезаписывает само значение переменной, а для расширенных типов он создает как-бы «псевдоним» присваиваемого объекта. Для того, чтобы именно перезаписать переменную новыми значениями, необходимо использовать функцию `copy()`.

```
a=[1,2,3]
b=a.copy()
b[0]=0
print(a)
print(b)
```

В результате на экране будет выведено [1,2,3] [0,2,3].

Вложенные списки (матрицы)

В расчетных задачах электроэнергетики часто требуются вычисления с матрицами. В Python нет встроенных типов данных для хранения и обработки матриц. Для этого используют вложенные списки. Любую матрицу можно рассматривать как «список списков». Рассмотрим пример матрицы размера 4 x 3:

```
matrix = [  
    [-1, 0, 1],  
    [-1, 0, 1],  
    [0, 1, -1],  
    [1, 1, -1]  
]
```

Либо в одну строку:

```
matrix = [[-1, 0, 1], [-1, 0, 1], [0, 1, -1], [1, 1, -1]]
```

Для вывода на экран можно воспользоваться уже известной функцией `print`.

Словари и кортежи

Помимо списков в языке Python существуют еще несколько удобных встроенных типов данных. К ним относятся словари и кортежи.

Словари (`dict`) – это одна из наиболее часто используемых структур данных, позволяющая хранить объекты, для доступа к которым используется ключ. Данные в словаре хранятся в формате `ключ – значение`. Если вспомнить такую структуру как список, то доступ к его элементам осуществляется по индексу, который представляет собой целое число. Причем индексы создаются автоматически, независимо от нас. В словаре аналогом индекса является ключ, при этом ответственность за его создание ложится на программиста.

Пустой словарь можно создать, используя функцию `dict()`, либо просто указав пустые фигурные скобки.

```
>>> d1 = dict()  
>>> print(type(d1))  
<class 'dict'>  
>>> d2 = {}  
>>> print(type(d2))  
<class 'dict'>
```

Если необходимо создать словарь с заранее подготовленным набором данных, то можно использовать один из перечисленных выше подходов, но с перечислением групп ключ-значение.

```
>>> d1 = dict(Ivan="manager", Mark="worker")
>>> print(d1)
{'Mark': 'worker', 'Ivan': 'manager'}
>>> d2 = {"A1": "123", "A2": "456"}
>>> print(d2)
{'A2': '456', 'A1': '123'}
```

Чтобы добавить элемент в словарь нужно указать новый ключ и значение.

```
>>> d1 = {"Russia": "Moscow", "USA": "Washington"}
>>> d1["China"] = "Beijing"
>>> print(d1)
{'Russia': 'Moscow', 'China': 'Beijing', 'USA': 'Washington'}
```

Для удаления элемента из словаря можно воспользоваться командой `del`, либо функцией словаря `pop()`. Пример:

```
>>> d2 = {"A1": "123", "A2": "456"}
>>> del d2["A1"]
>>> print(d2)
{'A2': '456'}
```

```
>>> d = {"A1": "123", "A2": "456"}
>>> d.pop("A1")
'123'
>>> print(d)
{'A2': '456'}
```

Как видно из примера, функция `pop` помимо удаления еще и в качестве результата выполнения возвращает значение удаленного элемента. В некоторых ситуациях это бывает полезно.

Доступ к элементу словаря, осуществляется так же как доступ к элементу списка, только в качестве индекса указывается ключ.

```
>>> d1 = {"Russia":"Moscow", "USA":"Washington"}
>>> d1["Russia"]
'Moscow'
```

Проверить наличие ключа в словаре можно с помощью оператора in:

```
>>> d2 = {"A1":"123", "A2":"456"}
>>> "A1" in d2
True
>>> "A3" in d2
False
```

У словарей доступен следующий набор дополнительных функций:

– **clear()**. Удаляет все элементы словаря. Пример:

```
>>> d2 = {"A1":"123", "A2":"456"}
>>> print(d2)
{'A2': '456', 'A1': '123'}
>>> d2.clear()
>>> print(d2)
{}
```

– **get(key,default=None)**. Возвращает значение из словаря по ключу key.

Если ключ не найден, возвращает переменную default. Пример:

```
>>> d = {"A1":"123", "A2":"456"}
>>> d.get("A1")
'123'
```

– **keys()**. Возвращает все ключи словаря.

```
>>> d = {"A1":"123", "A2":"456"}
>>> d.keys()
dict_keys(['A2', 'A1'])
```

Функция keys() может быть полезна, когда нужно со всеми ключами словаря провести какие-то действия (самый простой вариант – вывести на экран). В это случае функция keys() может быть использована в цикле

```
d = {"A1":"123", "A2":"456"}
for k in d.keys():
```

```
print(k)
```

На экране будут выведены строки «A1» и «A2».

– **values()**. Функция возвращает значения элементов словаря:

```
>>> d = {"A1": "123", "A2": "456"}
>>> d.values()
dict_values(['456', '123'])
```

Аналогично с функцией `keys()`, функция `values()` будет полезна, когда нужно произвести какие-то действия со всеми значениями словаря через цикл:

```
d = {"A1": "123", "A2": "456"}
for k in d.values():
    print(k)
```

В результате выполнения на экран будет выведены значения «123» и «456».

– **items()**. Возвращает элементы словаря (ключ, значение). По сути является комбинацией двух предыдущих функций.

Кортежи

В словарях в качестве ключа могут выступать практически любые типы данных, за некоторым исключением. Например, целые и дробные числа, строки, логические переменные могут выступать в качестве ключа, а вот списки в качестве ключа выступать не могут. Дело в том, что считается, что список в любой момент времени может измениться (в него могут быть добавлены новые, удалены старые элементы и так далее). Чтобы исправить эту ситуацию, в языке Python существует очень похожий на списки тип данных, который называется кортежем.

Кортеж — последовательность элементов, которые разделены между собой запятой и заключены в скобки. Это неизменяемый упорядоченный тип данных. Грубо говоря, кортеж - это список, который нельзя изменить. То есть, в кортеже есть только права на чтение. Поэтому элементы кортежа должны добавляться в него непосредственно в момент его создания.

Создаются кортежи почти точно так же, как и списки, только скобки ставятся не квадратные, а круглые.

Пример:

```
T = ('login', 'password')
```

Точно также, как строка может быть преобразована в число, а число в строку, ранее созданный список может быть преобразован в кортеж, а кортеж может быть преобразован в список. Для создания кортежа из списка существует функция `tuple()`, а для создания списка из кортежа существует функция `list()`:

```
sp=['a','b','c']  
tp=(1,2,3)  
sp2=list(tp)  
tp2=tuple(sp)
```

Для доступа к элементам кортежа точно также используются целочисленные индексы.

Для работы со списками доступны методы:

Метод	Что делает
list.append(x)	Добавляет элемент в конец списка
list.extend(L)	Расширяет список list, добавляя в конец все элементы списка L
list.insert(i, x)	Вставляет на i-ый элемент значение x
list.remove(x)	Удаляет первый элемент в списке, имеющий значение x. ValueError, если такого элемента не существует
list.pop([i])	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент

Метод	Что делает
list.index(x, [start [, end]])	Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end)
list.count(x)	Возвращает количество элементов со значением x
list.sort([key=функция])	Сортирует список на основе функции
list.reverse()	Разворачивает список
list.copy()	Поверхностная копия списка
list.clear()	Очищает список

Примеры решения задания

Задание 1.

Задан список элементов **lst = [1, 2, 3, 14, 33, 1, 1]**, необходимо проверить наличие элемента 1.

1.1 Конструкция **in**

	lst = [1, 2, 3, 14, 33, 1, 1]
	if 1 in lst:
	print('Есть')

Результат

Есть

1.2 Метод **count**

	lst = [1, 2, 3, 14, 33, 1, 1]
	if lst.count(1):

	<code>print('Есть')</code>
	<code>else:</code>
	<code>print('Нет')</code>

Результат

`Есть`

Задание 2.

Добавить в конец списка `lst = [1, 2, 3, 14, 33, 1, 1]` новые элементы 'Добавка' и 'Д', 'о', 'б', 'а', 'в', 'к', 'а'.

2.1 Метод **append()**

Данный метод добавляет в конец текущего списка новый элемент.

	<code>lst = [1, 2, 3, 14, 33, 1, 1]</code>
	<code>lst.append('Добавка')</code>
	<code>print(lst)</code>

Результат

`[1, 2, 3, 14, 33, 1, 1, 'Добавка']`

2.1 Метод **extend()**

Данный метод добавляет в конец текущего списка новые элементы в распакованном виде.

	<code>lst = [1, 2, 3, 14, 33, 1, 1]</code>
	<code>lst.extend('Добавка')</code>
	<code>print(lst)</code>

Результат

```
[1, 2, 3, 14, 33, 1, 1, 'д', 'о', 'б', 'а', 'в', 'к', 'а']
```

Задание 3.

Функция **slice()** – (срез) предназначена для представления части последовательности. Данная функция имеет специальное сокращение с указанием только трех параметров, разделенных знаком «:»

3.1 Изменить порядок элементов в списке.

	<pre>lst = [1, 2, 3, 14, 33, 1, 1]</pre>
	<pre>lst= lst[::-1]</pre>
	<pre>print(lst)</pre>

Результат

```
[1, 1, 33, 14, 3, 2, 1]
```

3.2 Вывести на печать каждый второй элемент списка.

	<pre>lst = [1, 2, 3, 14, 33, 1, 1]</pre>
	<pre>print(lst[::2])</pre>

Результат

```
[1, 3, 33, 1]
```

3.3 Вывести на печать элементы списка начиная с 3-го по 6-й (включительно) через один.

	<code>lst = [1, 2, 3, 14, 33, 1, 1]</code>
	<code>print(lst[slice(2, 6, 2)])</code>

Результат

`[3, 33]`

Задание для самостоятельного выполнения

Перед выполнением лабораторной работы каждый студент получает индивидуальное задание. Защита лабораторной работы происходит только после его выполнения (индивидуального задания). При защите лабораторной работы студент отвечает на контрольные вопросы, приведенные в конце, и поясняет выполненное индивидуальное задание. Ход защиты лабораторной работы контролируется преподавателем.

Задание

Задание 4.

Задайте список, состоящий из элементов 1, 2, 3, ... , 7, перечислением.

Задание 5.

Задайте список, состоящий из элементов 1, 2, 3, ... , 7, методом `append()`.

Задание 6.

Задайте список, состоящий из элементов 1, 2, 3, ... , 7, с помощью генератора списков.

Задание 7.

Вывести на печать элементы списка `lst = [1, 2, 3, 14, 33, 1, 1]` с нечетным индексом.

Задание 8.

Вывести на печать элементы списка `lst = [1, 2, 3, 14, 33, 1, 1]` с четным

индексом.

Задание 9.

Удалите все пустые элементы в списке `lst = [1, 2, 3, «», 14, 33, «», 1, 1]`.

Задание 10.

Удвойте значение каждого элемента списка `lst = [1, 2, 3, 14, 33, 1, 1]`.

Задание 11.

Сколько элементов будет содержать список `students` после следующих операций?

```
students = ['Ivan', 'Masha', 'Sasha']
```

```
students += ['Olga']
```

```
students += 'Olga'
```

Задание 12.

Имеется программа, код которой указан ниже.

Укажите, какие значения будут содержать списки в помеченных участках:

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

```
b = a
```

```
# значения списка b?
```

```
a[1] = 10
```

```
# значения списка b?
```

```
b[0] = 20
```

```
# значения списка a?
```

```
a = [5, 6]
```

```
# значения списка b?
```

Запишите значения списков в одну строку, разделяя списки точкой с запятой, а элементы внутри списка — пробелом, например:

```
1 1 1; 2 2 2; 3 3 3; 4 4 4
```

Контрольные вопросы

1. Перечислите характеристики типа данных список.
2. Как можно проверить наличие элемента в списке?
3. Чем отличаются методы `append()` и `extend()`?
4. Какие параметры можно передавать при срезах списков?
5. Какие способы создания списков из символов строки вы знаете?
6. Как создать список?
7. Какие операции со списками вы знаете?
8. Как получить доступ к отдельным элементам списка?
9. Что такое словари?
10. Что такое кортежи?
11. В чем отличие словарей, списков и кортежей?
12. Как добавлять элемент в словарь?
13. Как удалить элемент из словаря?
14. Как прочитать значение в словаре по ключу?

Практическое занятие №3

Тема: Циклы со счетчиком Принудительное завершение цикла

Цель работы: Получить основные сведения о циклах со счетчиком, особенностях их применения, перебираемых наборах данных, переменных цикла, дополнительных логических конструкциях.

Теоретическая часть

Цикл for

Другой тип циклов представляет конструкция for. Этот цикл пробегается по набору значений, помещает каждое значение в переменную, и затем в цикле мы можем с этой переменной производить различные действия. Формальное определение цикла for:

```
1  for переменная in набор_значений:
2      инструкции
```

После ключевого слова for идет название переменной, в которую будут помещаться значения. Затем после оператора in указывается набор значений и двоеточие. А со следующей строки располагается блок инструкций цикла, которые также должны иметь отступы от начала цикла. При выполнении цикла Python последовательно получает все значения из набора и передает их переменной. Когда все значения из набора будут перебраны, цикл завершает свою работу.

В качестве набора значений, например, можно рассматривать строку, которая по сути представляет набор символов. Посмотрим на примере:

```
1  message = "Hello"
2
3  for c in message:
```

```
4 | print(c)
```

В цикле определяется переменную `c`, после оператора `in` в качестве перебираемого набора указана переменная `message`, которая хранит строку "Hello". В итоге цикл `for` будет перебирать последовательно все символы из строки `message` и помещать их в переменную `c`. Блок самого цикла состоит из одной инструкции, которая выводит значение переменной `c` на консоль. Консольный вывод программы:

```
H
e
l
l
o
```

Цикл `for` также может иметь дополнительный блок `else`, который выполняется после завершения цикла:

```
1 | message = "Hello"
2 | for c in message:
3 |     print(c)
4 | else:
5 |     print(f"Последний символ: {c}. Цикл завершен");
6 | print("Работа программы завершена") # инструкция не имеет отступа,
   | поэтому не относится к else
```

В данном случае мы получим следующий консольный вывод:

```
H
e
l
l
o
Последний символ: o. Цикл завершен
Работа программы завершена
```

Стоит отметить, что блок `else` имеет доступ ко всем переменным, которые определены в цикле `for`.

Вложенные циклы. Одни циклы внутри себя могут содержать другие циклы. Рассмотрим на примере вывода таблицы умножения:

```
1  i = 1
2  j = 1
3  while i < 10:
4      while j < 10:
5          print(i * j, end="\t")
6          j += 1
7      print("\n")
8      j = 1
9      i += 1
```

Внешний цикл `while i < 10:` срабатывает 9 раз пока переменная `i` не станет равна 10. Внутри этого цикла срабатывает внутренний цикл `while j < 10:`. Внутренний цикл также срабатывает 9 раз пока переменная `j` не станет равна 10. Причем все 9 итераций внутреннего цикла срабатывают в рамках одной итерации внешнего цикла.

В каждой итерации внутреннего цикла на консоль будет выводиться произведение чисел `i` и `j`. Затем значение переменной `j` увеличивается на единицу. Когда внутренний цикл закончил работу, значений переменной `j` сбрасывается в 1, а значение переменной `i` увеличивается на единицу и происходит переход к следующей итерации внешнего цикла. И все повторяется, пока переменная `i` не станет равна 10. Соответственно внутренний цикл сработает всего 81 раз для всех итераций внешнего цикла. В итоге мы получим следующий консольный вывод:

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

2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Подобным образом можно определять вложенные циклы for:

```

1  for c1 in "ab":
2      for c2 in "ba":
3          print(f"{c1}{c2}")

```

В данном случае внешний цикл проходит по строке "ab" и каждый символ помещает в переменную c1. Внутренний цикл проходит по строке "ba", помещает каждый символ строки в переменную c2 и выводит сочетание обоих символов на консоль. То есть в итоге мы получим все возможные сочетания символов a и b:

```

ab
aa
bb
ba

```

Примеры решения задания

Задание 1.

14.1 Перебрать буквы в строке.

	<code>word = 'Слово'</code>
--	-----------------------------

	for letter in word:
	print(letter)

Результат

С
л
о
в
о

14.2 Перебрать элементы списка.

	shopping_list = ['milk', 'bread', 'cucumber', 'butter']
	for element in shopping_list:
	print(element)

Результат

milk
bread
cucumber
butter

Задание 2.

2.1. Перебрать буквы слова, букву «т» на печать не выводить.

	for letter in 'Стол':
	if letter == 'т':
	continue

	<code>print(letter)</code>
--	----------------------------

Результат

С
о
л

2.2. Перебрать буквы слова, после буквы «т» перебор окончить.

	<code>for letter in 'Стол':</code>
	<code>if letter == 'т':</code>
	<code>break</code>
	<code>print(letter)</code>

Результат

С

Задание для самостоятельного выполнения

Задание 3.

Создайте новый список, в котором содержатся только те элементы списка `lst[-11, 4, -2, 90, 400, 0, -5]`, значение по модулю которых больше 5.

Задание 4.

Исправьте нижеприведенный код программы так, чтобы на выходе не было заглавных букв.

```
letters = 'ЫГВЫОЯСремДШНККАыкШЙФа '
```

```
for letter in letters:
    if letter.upper() == letters:
        letters.replace(letter, '')
```

```
print(letters)
```

Задание 5.

Выведите на печать таблицу умножения в табличном виде.

Задание 6.

Выведите все числа от 10 до 25 включительно.

Задание 7.

Выведите все числа от 10 до 25 включительно в порядке убывания.

Задание 8.

Найдите факториал числа 8.

Задание 9.

Выведите на печать лесенку из 7 ступенек, i -я ступенька состоит из чисел от 1 до i .

Задание 10.

Создайте новый список, в котором содержатся только четные элементы списка `lst[-11, 4, -2, 90, 400, 0, -5]`.

Задание 11.

Выведите все элементы списка `lst[-11, 4, -2, 90, 400, 0, -5]`, которые больше предыдущего.

Задание 12.

Выведите первую пару элементов списка `lst[-11, 4, -2, 90, 400, 0, -5]`, которые имеют один знак.

Контрольные вопросы

1. Каково назначение циклов `for`?
2. Приведите особенности задания перебираемого набора?
3. Назначение дополнительного блока `else` в цикле `for`?
4. Что такое вложенные циклы и когда они применяются?
5. Охарактеризуйте операторы управления циклом.

Практическое занятие №4

Тема: Списки и циклы. Итерационный обход элементов списков.

Поиск элементов в списке. Циклы с предусловием. Циклы с постусловием. Пропуск итераций цикла.

Цель работы: Получить основные сведения о циклах с пред и пост условиями, синтаксис циклов с пред и пост условиями, условиях завершения цикла, бесконечных циклах, вложенных циклах.

Теоретическая часть

Цикл while

Цикл while проверяет истинность некоторого условия, и если условие истинно, то выполняет инструкции цикла. Он имеет следующее формальное определение:

```
1 while условие_выражение:
2     инструкции
```

После ключевого слова while указывается условие выражение, и пока это выражение возвращает значение True, будет выполняться блок инструкций, который располагается на последующих строках и должен иметь отступ от начала ключевого слова while.

```
1 number = 1
2
3 while number < 5:
4     print(f"number = {number}")
5     number += 1
6 print("Работа программы завершена")
```

В данном случае цикл `while` будет выполняться, пока переменная `number` меньше 5. Сам блок цикла состоит из двух инструкций (строки 4 и 5).

Весь процесс цикла можно представить следующим образом:

1. Сначала проверяется значение переменной `number` - меньше ли оно 5. И поскольку вначале переменная равна 1, то это условие возвращает `True`, и поэтому выполняются инструкции цикла. Инструкции цикла выводят на консоль строку `number = 1`. И далее значение переменной `number` увеличивается на единицу - теперь она равна 2. Однократное выполнение блока инструкций цикла называется итерацией. То есть таким образом, в цикле выполняется первая итерация.
2. Снова проверяется условие `number < 5`. Оно по прежнему равно `True`, так как `number = 2`, поэтому выполняются инструкции цикла. Инструкции цикла выводят на консоль строку `number = 2`. И далее значение переменной `number` опять увеличивается на единицу - теперь она равна 3. Таким образом, выполняется вторая итерация.
3. Опять проверяется условие `number < 5`. Оно по прежнему равно `True`, так как `number = 3`, поэтому выполняются инструкции цикла. Инструкции цикла выводят на консоль строку `number = 3`. И далее значение переменной `number` опять увеличивается на единицу - теперь она равна 4. То есть выполняется третья итерация.
4. Снова проверяется условие `number < 5`. Оно по прежнему равно `True`, так как `number = 4`, поэтому выполняются инструкции цикла. Инструкции цикла выводят на консоль строку `number = 4`. И далее значение переменной `number` опять увеличивается на единицу - теперь она равна 5. То есть выполняется четвертая итерация.

5. И вновь проверяется условие `number < 5`. Но теперь оно равно `False`, так как `number = 5`, поэтому выполняются выход из цикла. Все цикл - завершился. Далее уже выполняются действия, которые определены после цикла. Таким образом, данный цикл произведет четыре прохода или четыре итерации

В итоге при выполнении кода мы получим следующий консольный вывод:

```
number = 1
number = 2
number = 3
number = 4
Работа программы завершена
```

Для цикла `while` также можно определить дополнительный блок `else`, инструкции которого выполняются, когда условие равно `False`:

```
1  number = 1
2
3  while number < 5:
4      print(f"number = {number}")
5      number += 1
6  else:
7      print(f"number = {number}. Работа цикла завершена")
8  print("Работа программы завершена")
```

То есть в данном случае сначала проверяется условие и выполняются инструкции `while`. Затем, когда условие становится равным `False`, выполняются инструкции из блока `else`. Обратите внимание, что инструкции

из блока `else` также имеют отступы от начала конструкции цикла. В итоге в данном случае мы получим следующий консольный вывод:

```
number = 1
number = 2
number = 3
number = 4
number = 5. Работа цикла завершена
Работа программы завершена
```

Блок `else` может быть полезен, если условие изначально равно `False`, и мы можем выполнить некоторые действия по этому поводу:

```
1  number = 10
2
3  while number < 5:
4      print(f"number = {number}")
5      number += 1
6  else:
7      print(f"number = {number}. Работа цикла завершена")
8  print("Работа программы завершена")
```

В данном случае условие `number < 5` изначально равно `False`, поэтому цикл не выполняет ни одной итерации и сразу переходит в блок `else`.

Выход из цикла. Break и continue. Для управления циклом мы можем использовать специальные операторы `break` и `continue`. Оператор `break` осуществляет выход из цикла. А оператор `continue` выполняет переход к следующей итерации цикла. Оператор `break` может использоваться, если в цикле образуются условия, которые несовместимы с его дальнейшим выполнением. Рассмотрим следующий пример:

```
1  number = 0
```

```

2   while number < 5:
3       number += 1
4       if number == 3:    # если number = 3, выходим из цикла
5           break
6       print(f"number = {number}")

```

Здесь цикл `while` проверяет условие `number < 5`. И пока `number` не равно 5, предполагается, что значение `number` будет выводиться на консоль. Однако внутри цикла также проверяется другое условие: `if number == 3`. То есть, если значение `number` равно 3, то с помощью оператора `break` выходим из цикла. И в итоге мы получим следующий консольный вывод:

```

number = 1
number = 2

```

В отличие от оператора `break` оператор `continue` выполняет переход к следующей итерации цикла без его завершения. Например, в предыдущем примере заменим `break` на `continue`:

```

1   number = 0
2   while number < 5:
3       number += 1
4       if number == 3:    # если number = 3, переходим к новой
        итерации цикла
5           continue
6       print(f"number = {number}")

```

И в этом случае если значение переменной `number` равно 3, последующие инструкции после оператора `continue` не будут выполняться:

```

number = 1

```

```
number = 2
number = 4
number = 5
```

Примеры решения задания

Задание 1.

Например, можно написать программу, которая будет прибавлять к заданному числу 5 два до тех пор, пока получается число меньше 15.

	<code>i = 5</code>
	<code>while i < 15:</code>
	<code> print(i)</code>
	<code> i = i + 2</code>

Результат

5
7
9
11
13

Задание для самостоятельного выполнения

Задание 2.

Какой результат будет выведен на печать в результаты выполнения кода:

```
i=0
while True:
    i=i+2
    if i>=100:
```

```
break  
print(i)
```

Задание 3.

По заданному целому числу выведете на печать все квадраты натуральных чисел, не превосходящие заданное.

Задание 4.

Выведете наименьший натуральный делитель числа, отличный от 1.

Задание 5.

Определите наибольшую степень двойки не превосходящую заданное число.

Задание 6.

Для заданного числа определите, каким по счету оно является числом Фибоначи, если заданное число им не является на печать выведете -1.

Задание 7.

Для введенной последовательности 20 чисел, определите сколько ее элементов равны максимальному числу.

Задание 8.

Для введенной последовательности 20 чисел, определите сколько ее элементов больше предыдущего.

Задание 9.

Для введенной последовательности 20 чисел, определите наибольшее число подряд идущих элементов.

Задание 10.

Задано число a ($1 < a < 1.5$). Из чисел $1+1/2$, $1+1/3$, $1+1/4$..., напечатать те, которые не меньше a .

Задание 11.

Задано целое число k ($1 < k < 365$). Определить каким днем недели будет этот день. Год начинается в среду.

Контрольные вопросы

1. В чем основное отличие циклов for и while?
2. Приведите пример бесконечного цикла. Применяется ли он на практике?
3. Какой цикл используется если невозможно определить точное значение количества проходов?
4. Каково условие завершения цикла while?
5. Охарактеризуйте операторы управления циклом.

Практическое занятие №5

Тема: Функции. Создание функций. Параметры функции.

Возвращаемое значение. Вызов функции. Рекурсия. Функции с переменным числом параметров. Передача параметров по имени и по порядку. Значение параметров по умолчанию

Цель работы: Получить основные сведения о функциях в среде Python, синтаксис определения функций, возвращаемых значениях, рекурсии.

Теоретическая часть

Функции представляют блок кода, который выполняет определенную задачу и который можно повторно использовать в других частях программы. В предыдущих статьях уже использовались функции. В частности, функция `print()`, которая выводит некоторое значение на консоль. Python имеет множество встроенных функций и позволяет определять свои функции. Формальное определение функции:

```
1 def имя_функции ([параметры]) :  
2     инструкции
```

Определение функции начинается с выражения `def`, которое состоит из имени функции, набора скобок с параметрами и двоеточия. Параметры в скобках необязательны. А со следующей строки идет блок инструкций, которые выполняет функция. Все инструкции функции имеют отступы от начала строки.

Например, определение простейшей функции:

```
1 def say_hello() :  
2     print("Hello")
```

Функция называется `say_hello`. Она не имеет параметров и содержит одну единственную инструкцию, которая выводит на консоль строку "Hello".

Обратите внимание, что инструкции функции должны иметь отступы от начала функции. Например:

```
1 def say_hello():
2     print("Hello")
3
4
5 print("Bye")
```

Здесь инструкция `print("Bye")` не имеет отступов от начала функции `say_hello` и поэтому в эту функцию не входит. Обычно между определением функции и остальными инструкциями, которые не входят в функцию, располагаются две пустых строки.

Для вызова функции указывается имя функции, после которого в скобках идет передача значений для всех ее параметров:

```
1 имя_функции ([параметры])
```

Например, определим и вызовем функцию:

```
1 def say_hello():      # определение функции say_hello
2     print("Hello")
3
4
5 say_hello()           # вызов функции say_hello
6 say_hello()
7 say_hello()
```

Здесь три раза подряд вызывается функция `say_hello`. В итоге мы получим следующий консольный вывод:

```
Hello
Hello
Hello
```

Обратите внимание, что функция сначала определяется, а потом вызывается.

Если функция имеет одну инструкцию, то ее можно разместить на одной строке с остальным определением функции:

```
1  def say_hello(): print("Hello")
2
3  say_hello()
```

Подобным образом можно определять и вызывать и другие функции. Например, определим и выполним несколько функций:

```
1  def say_hello():
2      print("Hello")
3
4
5  def say_goodbye():
6      print("Good Bye")
7
8
9  say_hello()
10 say_goodbye()
```

Консольный вывод:

```
Hello
```

Локальные функции. Одни функции могут определяться внутри других функций - внутренние функции еще называют локальными. Локальные функции можно использовать только внутри той функции, в которой они определены. Например:

```
1  def print_messages():
2      # определение локальных функций
3      def say_hello(): print("Hello")
4      def say_goodbye(): print("Good Bye")
5      # вызов локальных функций
6      say_hello()
7      say_goodbye()
8
9  # Вызов функции print_messages
10 print_messages()
11
12 #say_hello() #
    вне функции print_messages функция say_hello не доступна
```

Здесь функции `say_hello()` и `say_goodbye()` определены внутри функции `print_messages()` и поэтому по отношению к ней являются локальными. Соответственно они могут использоваться только внутри функции `print_messages()`.

Организация программы и функция `main`. В программе может быть определено множество функций. И чтобы всех их упорядочить, одним из способов их организации является добавление специальной функции (обычно называется `main`), в которой потом уже вызываются другие функции:

```

1  def main():
2      say_hello()
3      say_goodbye()
4
5  def say_hello():
6      print("Hello")
7
8  def say_goodbye():
9      print("Good Bye")
10
11 # Вызов функции main
12 main()

```

Параметры функции. Функция может принимать параметры. Через параметры в функцию можно передавать данные. Банальный пример - функция `print()`, которая с помощью параметра принимает значение, выводимое на консоль. Теперь определим и используем свою функцию с параметрами:

```

1  def say_hello(name):
2      print(f"Hello, {name}")
3
4  say_hello("Tom")
5  say_hello("Bob")
6  say_hello("Alice")

```

Функция `say_hello` имеет параметр `name`, и при вызове функции мы можем передать этому параметру какой-либо значение. Внутри функции мы

можем использовать параметр как обычную переменную, например, вывести значение этого параметра на консоль функцией `print`. Так, в выражении:

```
1 | say_hello("Tom")
```

Строка "Tom" будет передаваться параметру `name`. В итоге при выполнении программы мы получим следующий консольный вывод:

```
Hello, Tom
Hello, Bob
Hello, Alice
```

При вызове функции значения передаются параметрам по позиции. Например, определим и вызовем функцию с несколькими параметрами:

```
1 | def print_person(name, age):
2 |     print(f"Name: {name}")
3 |     print(f"Age: {age}")
4 |
5 |
6 | print_person("Tom", 37)
```

Здесь функция `print_person` принимает два параметра: `name` и `age`. При вызове функции:

```
1 | print_person("Tom", 37)
```

Первое значение - "Tom" передается первому параметру, то есть параметру `name`. Второе значение - 37 передается второму параметру - `age`. И внутри функции значения параметров выводятся на консоль:

```
Name: Tom
```

```
Age: 37
```

Значения по умолчанию. Некоторые параметры функции мы можем сделать необязательными, указав для них значения по умолчанию при определении функции. Например:

```
1 def say_hello(name="Tom") :
2     print(f"Hello, {name}")
3
4
5 say_hello()          # здесь параметр name будет иметь
   значение "Tom"
6 say_hello("Bob")     # здесь name = "Bob"
```

Здесь параметр `name` является необязательным. И если мы не передаем при вызове функции для него значение, то применяется значение по умолчанию, то есть строка `"Tom"`. Консольный вывод данной программы:

```
Hello, Tom
```

```
Hello, Bob
```

Если функция имеет несколько параметров, то необязательные параметры должны идти после обязательных. Например:

```
1 def print_person(name, age = 18):
2     print(f"Name: {name} Age: {age}")
3
4
5 print_person("Bob")
```

```
6 | print_person("Tom", 37)
```

Здесь параметр `age` является необязательным и по умолчанию имеет значение 18. Перед ним расположен обязательный параметр `name`. Поэтому при вызове функции мы можем не передавать значение параметру `age`, но параметру `name` передать значение необходимо.

При необходимости мы можем сделать все параметры необязательными:

```
1 | def print_person(name = "Tom", age = 18):
2 |     print(f"Name: {name} Age: {age}")
3 |
4 |
5 | print_person()           # Name: Tom Age: 18
6 | print_person("Bob")     # Name: Bob Age: 18
```

Передача значений параметрам по имени. Именованные параметры. В примерах выше при вызове функции значения передаются параметрами функции по позиции. Но также можно передавать значения параметрам по имени. Для этого при вызове функции указывается имя параметра и ему присваивается значение:

```
1 | def print_person(name, age):
2 |     print(f"Name: {name} Age: {age}")
3 |
4 |
5 | print_person(age = 22, name = "Tom")
```

В данном случае значения параметрам `age` и `name` передаются по имени. И несмотря на то, что параметр `name` идет первым в определении

функции, мы можем при вызове функции написать `print_person(age = 22, name = "Tom")` и таким образом передать число 22 параметру `age`, а строку "Tom" параметру `name`.

Символ `*` позволяет установить, какие параметры будут именованными - то есть такие параметры, которым можно передать значения только по имени. Все параметры, которые располагаются справа от символа `*`, получают значения только по имени:

```
1 def print_person(name, *, age, company):
2     print(f"Name: {name} Age: {age} Company: {company}")
3
4
5 print_person("Bob", age = 41, company = "Microsoft")    #
   Name: Bob Age: 41 company: Microsoft
```

В данном случае параметры `age` и `company` являются именованными.

Можно сделать все параметры именованными, поставив перед списком параметров символ `*`:

```
1 def print_person(*, name, age, company):
2     print(f"Name: {name} Age: {age} Company: {company}")
```

Если наоборот надо определить параметры, которым можно передавать значения только по позиции, то есть позиционные параметры, то можно использовать символ `/`: все параметры, которые идут до символа `/`, являются позиционными и могут получать значения только по позиции

```
1 def print_person(name, /, age, company="Microsoft"):
2     print(f"Name: {name} Age: {age} Company: {company}")
3
```

```

4
5 print_person("Tom", company="JetBrains", age = 24)      #
  Name: Tom Age: 24 company: JetBrains
6 print_person("Bob", 41)                                # Name: Bob Age:
  41 company: Microsoft

```

В данном случае параметр `name` является позиционным.

Для одной функции можно определять одновременно позиционные и именованные параметры.

```

1 def print_person(name, /, age = 18, *, company):
2     print(f"Name: {name} Age: {age} Company: {company}")
3
4
5 print_person("Sam", company = "Google")                #
  Name: Sam Age: 18 company: Google
6 print_person("Tom", 37, company = "JetBrains")         #
  Name: Tom Age: 37 company: JetBrains
7 print_person("Bob", company = "Microsoft", age = 42)  #
  Name: Bob Age: 42 company: Microsoft

```

В данном случае параметр `name` располагается слева от символа `/`, поэтому является позиционным и обязательным - ему можно передать значение только по позиции.

Параметр `company` является именованным, так как располагается справа от символа `*`. Параметр `age` может получать значение по имени и по позиции.

Неопределенное количество параметров. С помощью символа звездочки можно определить параметр, через который можно передавать неопределенное количество значений. Это может быть полезно, когда мы

хотим, чтобы функция получала несколько значений, но мы точно не знаем, сколько именно. Например, определим функцию подсчета суммы чисел:

```
1 def sum(*numbers):
2     result = 0
3     for n in numbers:
4         result += n
5     print(f"sum = {result}")
6
7
8 sum(1, 2, 3, 4, 5)      # sum = 15
9 sum(3, 4, 5, 6)        # sum = 18
```

В данном случае функция `sum` принимает один параметр - `*numbers`, но звездочка перед названием параметра указывает, что фактически на место этого параметра мы можем передать неопределенное количество значений или набор значений. В самой функции с помощью цикла `for` можно пройтись по этому набору, получить каждое значение из этого набора в переменную `n` и произвести с ним какие-нибудь действия. Например, в данном случае вычисляется сумма переданных чисел.

Возвращение результата. Функция может возвращать результат. Для этого в функции используется оператор `return`, после которого указывается возвращаемое значение:

```
1 def имя_функции ([параметры]):
2     инструкции
3     return возвращаемое_значение
```

Определим простейшую функцию, которая возвращает значение:

```
1 def get_message():
2     return "Hello"
```

Здесь после оператора `return` идет строка `"Hello "` - это значение и будет возвращать функция `get_message()`. Затем это результат функции можно присвоить переменной или использовать как обычное значение:

```
1 def get_message():
2     return "Hello"
3
4
5 message = get_message() # получаем результат функции
   get_message в переменную message
6 print(message)          # Hello
7
8 # можно напрямую передать результат функции get_message
9 print(get_message())    # Hello
```

После оператора `return` может идти и сложное вычисляемое выражение, результат которого будет возвращаться из функции. Например, определим функцию, которая увеличивает число в два раза:

```
1 def double(number):
2     return 2 * number
```

Здесь функция `double` будет возвращать результат выражения `2 * number`:

```
1 def double(number):
2     return 2 * number
```

```

3
4     result1 = double(4)          # result1 = 8
5     result2 = double(5)          # result2 = 10
6     print(f"result1 = {result1}") # result1 = 8
7     print(f"result2 = {result2}") # result2 = 10

```

Или другой пример - получение суммы чисел:

```

1     def sum(a, b):
2         return a + b
3
4
5     result = sum(4, 6)             # result = 0
6     print(f"sum(4, 6) = {result}") # sum(4, 6) = 10
7     print(f"sum(3, 5) = {sum(3, 5)}") # sum(3, 5) = 8

```

Выход из функции. Оператор `return` не только возвращает значение, но и производит выход из функции. Поэтому он должен определяться после остальных инструкций. Например:

```

1     def get_message():
2         return "Hello"
3         print("End of the function")
4
5     print(get_message())

```

С точки зрения синтаксиса данная функция корректна, однако ее инструкция `print("End of the function")` не имеет смысла - она никогда не выполнится, так как до ее выполнения оператор `return` возвратит значение и произведет выход из функции.

Однако мы можем использовать оператор `return` и в таких функциях, которые не возвращают никакого значения. В этом случае после оператора `return` не ставится никакого возвращаемого значения. Типичная ситуация - в зависимости от определенных условий произвести выход из функции:

```
1  def print_person(name, age):
2      if age > 120 or age < 1:
3          print("Invalid age")
4          return
5      print(f"Name: {name}  Age: {age}")
6
7
8  print_person("Tom", 22)
9  print_person("Bob", -102)
```

Здесь функция `print_person` в качестве параметров принимает имя и возраст пользователя. Однако в функции вначале мы проверяем, соответствует ли возраст некоторому диапазону (меньше 120 и больше 0). Если возраст находится вне этого диапазона, то выводим сообщение о недопустимом возрасте и с помощью оператора `return` выходим из функции. После этого функция заканчивает свою работу.

Однако если возраст корректен, то выводим информацию о пользователе на консоль. Консольный вывод:

```
Name: Tom  Age: 22
Invalid age
```

Функция как тип. В Python функция фактически представляет отдельный тип. Так мы можем присвоить переменной какую-нибудь

функцию и затем, используя переменную, вызывать данную функцию.

Например:

```
1 def say_hello(): print("Hello")
2 def say_goodbye(): print("Good Bye")
3
4 message = say_hello
5 message()          # Hello
6 message = say_goodbye
7 message()          # Good Bye
```

В данном случае переменной `message` присваивается одна из функций. Сначала ей передается функция `say_hello()`:

```
1 message = say_hello
```

После этого переменная `message` будет указывать на данную функцию, то есть фактически представлять функцию `say_hello`. А это значит, что мы можем вызывать переменную `message` как обычную функцию:

```
1 message()          # Hello
```

Фактически это приведет к выполнению функции `say_hello`, и на консоль будет выведена строка "Hello". Затем подобным образом мы можем передать переменной `message` другую функцию и вызвать ее.

Подобным образом можно через переменную вызывать функцию с параметрами и возвращать ее результат:

```
1 def sum(a, b): return a + b
2 def multiply(a, b): return a * b
```

```

3
4 operation = sum
5 result = operation(5, 6)
6 print(result)    # 11
7
8 operation = multiply
9 print(operation(5, 6))    # 30

```

Функция как параметр функции. Поскольку функция в Python может представлять такое же значение как строка или число, соответственно мы можем передать ее в качестве параметра в другую функцию. Например, определим функцию, которая выводит на консоль результат некоторой операции:

```

1 def do_operation(a, b, operation):
2     result = operation(a, b)
3     print(f"result = {result}")
4
5 def sum(a, b): return a + b
6 def multiply(a, b): return a * b
7
8 do_operation(5, 4, sum)    # result = 9
9 do_operation(5, 4, multiply)    # result = 20

```

В данном случае функция `do_operation` имеет три параметра, причем третий параметр, как предполагается, будет представлять функцию, которая принимает два параметра и возвращает некоторый результат. Иными словами третий параметр - `operation` представляет некоторую операцию, но на момент определения функции `do_operation` мы точно не знаем, что это

будет за операция. Мы только знаем, что она принимает два параметр и возвращает какой-то результат, который потом выводится на консоль.

При вызове функции `do_operation` мы сможем передать в качестве третьего параметра другую функцию, например, функцию `sum`:

```
1 | do_operation(5, 4, sum)
```

То есть в данном случае параметр `operation` фактически будет представлять функцию `sum` и будет возвращать сумму двух чисел.

Затем аналогичным образом в вызов функции `do_operation` можно передать третьему параметру другую функцию - `multiply`, которая выполнит умножение чисел:

```
1 | do_operation(5, 4, multiply)    # result = 20
```

Таким образом, более гибкие по функциональности функции, которые через параметры принимают другие функции.

Функция как результат функции. Также одна функция в Python может возвращать другую функцию. Например, определим функцию, которая в зависимости от значения параметра возвращает ту или иную операцию:

```
1 | def sum(a, b): return a + b
2 | def subtract(a, b): return a - b
3 | def multiply(a, b): return a * b
4 |
5 |
6 | def select_operation(choice):
7 |     if choice == 1:
8 |         return sum
```

```

9         elif choice == 2:
10             return subtract
11         else:
12             return multiply
13
14
15 operation = select_operation(1)           # operation = sum
16 print(operation(10, 6))                  # 16
17
18 operation = select_operation(2)           # operation = subtract
19 print(operation(10, 6))                  # 4
20
21 operation = select_operation(3)           # operation = multiply
22 print(operation(10, 6))                  # 60

```

В данном случае функция `select_operation` в зависимости от значения параметра `choice` возвращает одну из трех функций - `sum`, `subtract` и `multiply`. Затем мы можем получить результат функции `select_operation` в переменную `operation`:

```

1 operation = select_operation(1)

```

Так, в данном случае в функцию `select_operation` передается число 1, соответственно она будет возвращать функцию `sum`. Поэтому переменная `operation` фактически будет указывать на функцию `sum`, которая выполняет сложение двух чисел:

```

1 print(operation(10, 6))                  # 16 -
   фактически равно sum(10, 6)

```

Аналогичным образом можно получить и выполнить другие функции.

Лямбда-выражения в языке Python представляют небольшие анонимные функции, которые определяются с помощью оператора `lambda`. Формальное определение лямбда-выражения:

```
1 lambda [параметры] :  
    инструкция
```

Определим простейшее лямбда-выражение:

```
1 message = lambda: print("hello")  
2  
3 message()    # hello
```

Здесь лямбда-выражение присваивается переменной `message`. Это лямбда-выражение не имеет параметров, ничего не возвращает и просто выводит строку "hello" на консоль. И через переменную `message` мы можем вызвать это лямбда-выражение как обычную функцию. Фактически оно аналогично следующей функции:

```
1 def message():  
2     print("hello")
```

Если лямбда-выражение имеет параметры, то они определяются после ключевого слова `lambda`. Если лямбда-выражение возвращает какой-то результат, то он указывается после двоеточия. Например, определим лямбда-выражение, которое возвращает квадрат числа:

```
1 square = lambda n: n * n  
2
```

```
3 print(square(4))      # 16
4 print(square(5))      # 25
```

В данном случае лямбда-выражение принимает один параметр - n . Справа от двоеточия идет возвращаемое значение - $n * n$. Это лямбда-выражение аналогично следующей функции:

```
1 def square2(n): return n * n
```

Аналогичным образом можно создавать лямбда-выражения, которые принимают несколько параметров:

```
1 sum = lambda a, b: a + b
2
3 print(sum(4, 5))      # 9
4 print(sum(5, 6))      # 11
```

Хотя лямбда-выражения позволяют немного сократить определения функций, тем не менее они ограничены тем, что они могут выполнять только одно выражение. Однако они могут быть довольно удобны в тех случаях, когда необходимо использовать функцию для передачи в качестве параметра или возвращения в другой функции. Например, передача лямбда-выражения в качестве параметра:

```
1 def do_operation(a, b, operation):
2     result = operation(a, b)
3     print(f"result = {result}")
4
5 do_operation(5, 4, lambda a, b: a + b) # result = 9
6 do_operation(5, 4, lambda a, b: a * b) # result = 20
```

В данном случае нам нет необходимости определять функции, чтобы передать их в качестве параметра.

То же самое касается и возвращение лямбда-выражений из функций:

```
1  def select_operation(choice):
2      if choice == 1:
3          return lambda a, b: a + b
4      elif choice == 2:
5          return lambda a, b: a - b
6      else:
7          return lambda a, b: a * b
8
9
10 operation = select_operation(1) # operation = sum
11 print(operation(10, 6)) # 16
12
13 operation = select_operation(2) # operation = subtract
14 print(operation(10, 6)) # 4
15
16 operation = select_operation(3) # operation = multiply
17 print(operation(10, 6)) # 60
```

Рекурсия в программировании — это возможность дать определение функции, используя в процессе саму определяемую функцию. В математике многие функции определены именно таким образом, поэтому и большинство языков программирования используют этот подход.

Это работает и в Python. Обычно в определении функции можно использовать только определения, которые дали ранее. Но есть одно

исключение — функция в своем теле может вызывать себя. Выглядит это так:

```
1  def factorial(n):  
2      if n <= 0:  
3          return 1  
4      return n * factorial(n - 1)
```

В примере выше используется условие, которое прекращает рекурсию. Если в этой функции убрать условие, которое проверяет аргумент на неотрицательность, то первый же вызов этой функции заставит программу заиклиться — функция продолжит вызывать себя постоянно.

В определениях рекурсивных функций практически всегда есть подобное условие. Оно позволяет вычислению пойти по одной из веток:

По рекурсивной — в этой ветке произойдет вызов себя;

По терминальной — закончит вычисление и вернет результат.

Какой-то из аргументов рекурсивной функции должен обязательно убывать. В качестве убывания может быть:

Уменьшение счетчика;

Отбрасывание головы списка при движении к его хвосту;

Вызов себя для части исходной структуры при обработке древовидных структур данных.

Чтобы понять, что программа не заиклится, используют метод «пристального взгляда» и тесты. Особенно важно проверять срабатывание условия завершения рекурсии.

В большинстве программ, написанных на поддерживающих вызов функции языках, этот вызов устроен так: перед вызовом функции текущее место в программе запоминается в стеке. А когда функция возвращает результат, то соответствующий элемент стека отбрасывается.

Стек (stack) — это абстрактный тип данных, который похож на стопку монет. Монета, которую положили последней, будет снята первой. И при снятии монет порядок получается обратным порядку складывания.

В этом же стеке сохраняются значения аргументов функции, а иногда и другая служебная информация. При этом память, которая выделяется для стека при запуске программы, конечна и ограничена.

Если функция будет вызывать себя постоянно и не возвращать результат, то память в итоге закончится. Когда заканчивается память, выделенная для стека вызовов, стек переполняется.

В итоге мы не сможем посчитать факториал для достаточно больших чисел с помощью рекурсивной функции. Но сможем посчитать с помощью итеративной — написанной с использованием циклов и переменных.

Некоторые алгоритмы реализуются проще, если использовать именно рекурсию, а не циклы. Часто такие алгоритмы работают с рекурсивными структурами данных — деревьями, «словарями словарей словарей» и подобными. При реализации таких алгоритмов нужно помнить, что память для стека конечна. При этом обычно конечны и сами обрабатываемые структуры данных, поэтому отказываться полностью от рекурсии не стоит.

Существует несколько видов рекурсии:

- Прямая — когда функция вызывает себя;

- Косвенная — когда одна функция внутри себя вызывает другую функцию, которая когда-то вызовет первую;

- Линейная — когда при вычислении результата функции функция вызывает себя один раз, как в примере с `factorial`. Уточним, что «один раз» — это не про общее количество вызовов функции в теле. Речь идет о количестве вызовов, результаты которых нужны для одного общего вычисления;

- Каскадная — когда функция вызывает себя несколько раз.

Примеры решения задания

Задание 1.

1.1 Напишите простейшую функцию печати двух чисел:

```
def my_func(a, b=6):  
    print(a, b)  
  
# b=6 - аргумент по умолчанию  
# Тесты  
my_func(2, 4) # Позиционные аргументы  
my_func(b=2, a=4) # Ключевые аргументы  
my_func(2, b=4) # Первый аргумент - позиционный, второй - ключевой  
my_func(a=2, 4) # Позиционные аргументы должны всегда предшествовать  
ключевым, поэтому в работе кода возникнет ошибка
```

Результат

```
2 4  
4 2  
2 4  
SyntaxError: positional argument follows keyword argument
```

1.2 Напишите простейшую функцию печати любого количества аргументов:

```
def my_func(*args):  
    print(args)  
  
# Тесты  
my_func(2, 4)  
my_func()  
my_func(2, 4, key=10)
```

Результат

```
(2, 4)  
()
```

```
TypeError: my_func() got an unexpected keyword argument 'key'
```

1.3 Напишите простейшую функцию печати любого количества ключевых аргументов:

```
def my_func(**kwargs):  
    print(kwargs)  
  
# Тесты  
my_func()  
my_func(a=5, b=10, c=15)  
my_func(a=5, b=10, 8)
```

Результат

```
{}  
{'a': 5, 'b': 10, 'c': 15}  
SyntaxError: positional argument follows keyword argument
```

Задание для самостоятельного выполнения

Задание 2.

Напишите функцию **sum_range(start, end)**, которая суммирует все целые числа от значения **start** до величины **end** включительно. Если пользователь задаст первое число большее чем второе, просто поменяйте их местами.

Задание 3.

Создайте функцию **three_args()**, которая принимает **1, 2** или **3** строго ключевых параметра. В результате ее работы на печать в консоль выводятся значения переданных переменных, но только если они не равны **None**. Получим, например, следующее сообщение: **Переданы аргументы: var1 = 2, var3 = 10**.

Задание 4.

Напишите функцию `capitalize()`, которая принимает слово из маленьких латинских букв и возвращает его же, меняя первую букву на большую.

Задание 5.

Дано действительное положительное число a и целое неотрицательное число n . Вычислите a^n используя рекуррентное соотношение $a^n = a \cdot a^{n-1}$. Решение оформите в виде функции `power(a, n)`.

Задание 6.

Дана последовательность целых чисел, заканчивающаяся числом 0. Выведите эту последовательность в обратном порядке. Решите задачу с помощью рекурсии.

Задание 7.

Написать функцию `is_year_leap`, принимающую 1 аргумент — год, и возвращающую `True`, если год високосный, и `False` иначе.

Задание 8.

Написать функцию `season`, принимающую 1 аргумент — номер месяца (от 1 до 12), и возвращающую время года, которому этот месяц принадлежит (зима, весна, лето или осень).

Задание 9.

Написать функцию `is_prime`, принимающую 1 аргумент — число от 0 до 1000, и возвращающую `True`, если оно простое, и `False` - иначе.

Задание 10.

Напишите функцию `personalData`, которая принимая аргументы имеет соответствующий вывод:

```
personalData(first_name='John', last_name='Doe', age=28, position='Python
developer')
```

```
age: 28
```

```
first_name: John
```

```
last_name: Doe
```

```
position: Python developer
```

```
personalData(first_name='Jack', last_name='Smith', age=32, work_experience =  
'5 years', position='Project manager')
```

```
age: 32  
first_name: Jack  
last_name: Smith  
position: Project manager  
work_experience: 5 years
```

Задание 11.

Напишите функцию **evaluate(coefficients, x)**, которая принимает список коэффициентов и значение аргумента **x**.

Контрольные вопросы

1. Что такое функция в языке? Что такое аргумент функции?
2. Что такое переменная? Чем характеризуется переменная?
3. Что такое оператор? Какие операторы вы знаете?
4. Как выводить на экран значения переменных?
5. Как ввести с клавиатуры значения переменных?

Практическое занятие №6

Тема: Библиотека Matplotlib. Подключение библиотеки. Создание линейного графика. Создание круговой диаграммы. Создание столбчатой диаграммы. Настройки отображения

Цель работы: Получить основные сведения о библиотеке **Matplotlib**, элементом построения графика Figure, ее составляющими, управлении осями, легендой и сеткой, возможностями построения нескольких графиков.

Теоретическая часть

Язык Python обрел свою популярность в том числе благодаря множеству удобных библиотек для визуализации данных. Одной из наиболее распространенных таких библиотек является библиотека Matplotlib. Для ее установки необходимо через командную строку операционной системы выполнить следующую команду:

```
pip install matplotlib
```

При выполнении этой команды будет автоматически скачана последняя версия библиотеки и записана в нужный каталог системы. Далее необходимо импортировать функции для работы с графикой. Библиотека Matplotlib имеет несколько модулей. Практически все задачи, связанные с построением графиков, можно решить, используя возможности, которые предоставляет модуль pyplot. Для его подключения в программу его необходимо импортировать, используя соответствующую команду:

```
import matplotlib.pyplot as plt
```

Здесь мы задаем псевдоним plt, который позволяет сократить объем кода и сделать его более читаемым. Вместо слова plt вы можете придумать любой другой псевдоним.

Библиотека Matplotlib имеет огромное количество функций и их параметров. Рассмотреть все возможности библиотеки в рамках одной лекции не предоставляется возможным. Целью лекции является получение общих представлений об устройстве библиотеки и ее основных функциях, достаточных для дальнейшего ее самостоятельного изучения с помощью официальной документации.

Рассмотрим основные термины и понятия, касающиеся изображения графика, с которыми вам необходимо будет познакомиться, для того, чтобы в дальнейшем у вас не было трудностей при работе с документацией по библиотеке matplotlib.

Корневым элементом при построения графиков в системе Matplotlib является Фигура (Figure). Все, что нарисовано на рисунке является элементами фигуры. Рассмотрим ее составляющие более подробно.

График

На рисунке представлены два графика – линейный и точечный. Matplotlib предоставляет огромное количество различных настроек, которые можно использовать для того, чтобы придать графику вид, который вам нужен: цвет, толщина и тип линии, стиль линии и многое другое, все это вы можете найти в документации к библиотеке.

Рисунок 1 – Основные графические элементы библиотеки Matplotlib

Оси

Вторым, после непосредственно самого графика, по важности элементом фигуры являются оси. Для каждой оси можно задать метку (подпись), основные (major) и дополнительные (minor) тики, их подписи, размер и толщину, также можно задать диапазоны по каждой из осей.

Сетка и легенда

Следующими элементами фигуры, которые значительно повышают информативность графика являются сетка и легенда. Сетка также может быть основной (major) и дополнительной (minor). Каждому типу сетки можно задавать цвет, толщину линии и тип. Для отображения сетки и легенды используются соответствующие команды.

Основным элементом изображения, которое строит pyplot является Фигура (Figure), на нее накладываются один или более графиков, осей, надписей и т.п. Для построения графика используется команда plot(). В самом минимальном варианте можно ее использовать без параметров:

```
import matplotlib.pyplot as plt
plt.plot()
plt.show()
```

Первая функция (plot) строит график, вторая функция (show) отображает его на экране.

В результате выполнения кода будет выведено пустое поле:

Рисунок 2 – Построение пустого графика

Если в качестве параметра функции plot() передать список, то значения из этого списка будут отложены по оси ординат (ось y), а по оси абсцисс (ось x) будут отложены индексы элементов массива:

```
plt.plot([1, 7, 3, 5, 11, 1])
```

Рисунок 3 – Построение графика на основе списка

Для того, чтобы задать значения по осям x и y необходимо в plot() передать два списка:

```
plt.plot([1, 5, 10, 15, 20], [1, 7, 3, 5, 11])
```

В результате получим следующий график:

Рисунок 4 – Построение графика на основе двух списков

Таким образом, если нам нужно вывести какие-то данные, очевидно, нам нужно предварительно их подготовить. Например, если мы хотим построить график синусоиды, можно использовать рассмотренные ранее циклы и словари:

```
import math
import matplotlib.pyplot as plt
a=dict()
for i in range(0, 360):
    a[i]=math.sin(i/180*math.pi)

plt.plot(a.keys(),a.values())
plt.show()
```

В результате будет создан следующий график:

Рисунок 5 – Пример построение графика синусоиды

Здесь встречается не рассмотренная ранее библиотека `math`. Она содержит множество наиболее часто используемых математических функций, в том числе необходимую нам тригонометрическую функцию синуса. Список всех доступных функций библиотеки `math` можно найти в документации на официальном сайте проекта Python.

Текстовые надписи на графике

Наиболее часто используемые текстовые надписи на графике это:

- наименование осей;
- наименование самого графика;
- текстовое примечание на поле с графиком;
- легенда.

Рассмотрим кратко данные элементы.

Для задания подписи оси x используется функция **xlabel**, оси y – **ylabel**. Разберемся с аргументами данных функций.

Аргументов у этих функций довольно много и они позволяют достаточно тонко настроить внешний вид надписей. В рамках настоящей работы мы рассмотрим только основные из них. Полный список всегда можно посмотреть в официальной документации. Для функций **xlabel/ylabel** основными являются следующие аргументы:

- текст подписи: **str** (первый аргумент);
- расстояние между областью графика, включающую оси, и меткой: **labelpad** (численное значение, по умолчанию отсутствует).

Функции **xlabel/ylabel** принимают в качестве аргументов параметры класса `matplotlib.text.Text`, некоторые из них нам могут пригодиться:

- размер шрифта: **fontsize** или **size** – число либо значение из списка: {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}.
- стиль шрифта: **fontstyle** – значение из списка: {'normal', 'italic', 'oblique'};
- толщина шрифта: **fontweight** – число в диапазоне от 0 до 1000 либо значение из списка: {'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'}.

Цвет шрифта – **color**: один из доступных способов определения цвета (см. Цвет линии в официальной документации).

Пример использования:

```
plt.xlabel('Day', fontsize=15, color='blue')
```

Заголовок графика

Для задания заголовка графика используется функция **title**:

```
plt.title('Chart price', fontsize=17)
```

Из параметров отметим следующие:

- текст заголовка (первый параметр, **label**, тип - str).
- выравнивание заголовка: **loc**, значение из набора {'center', 'left', 'right'}

Для функции **title** также доступны параметры конструктора класса `matplotlib.text.Text`, часть из которых рассмотрены в описании аргументов функций **xlabel** / **ylabel**.

Текстовое примечание

За размещение текста на поле графика отвечает функция **text**, которой вначале передаются координаты позиции надписи, после этого – текст самой надписи.

```
plt.text(1, 1, '&apos;type: Steel&apos;')
```

Легенда

Легенда будет размещена на графике, если вызвать функцию `legend()`. Функция позволяет настроить расположение легенды, размер шрифта и многие другие параметры. В рамках настоящей работы мы не будем рассматривать аргументы функции, при желании вы можете изучить их самостоятельно.

Давайте немного наведем красоту на наших графиках, используя рассмотренные функции:

```
import math
import matplotlib.pyplot as plt
a=dict()

for i in range(0, 360):
    a[i]=math.sin(i/180*math.pi)

plt.title("y = sin(x)") # заголовок
plt.xlabel("x") # ось абсцисс
plt.ylabel("y") # ось ординат
plt.grid() # включение отображение сетки
```

```
plt.plot(a.keys(), a.values())  
plt.legend(['&apos;Синус&apos;'])  
plt.show()
```

Результат работы кода:

Рисунок 6 – Настройка параметров графика

Помимо рассмотренных функций, здесь мы использовали еще одну функцию **grid**. Она позволяет отобразить сетку на графике. Функция позволяет настроить цвет сетки, стиль линии, их толщину, шаг сетки, отображения по осям и так далее.

Например:

```
plt.grid(color='r', linestyle='--',  
&apos;, linewidth=2)
```

нарисует следующую сетку:

Рисунок 7 – Настройка параметров сетки

Список всех параметров функции `grid` и описание множества других полезных функции пакета `matplotlib.pyplot` можно найти в документации на официальном сайте библиотеки `matplotlib`.

Кстати функция `plot` тоже позволяет настраивать цвет, толщину и стиль линии с помощью этих же параметров (`color`, `linestyle`, `linewidth`).

Построение одновременно нескольких графиков

Если нам нужно построить два графика на одном листе, мы просто вызываем второй раз функцию `plot`.

```
import math  
import matplotlib.pyplot as plt
```

```

a=dict()
b=dict()
for i in range(0, 360):
    a[i]=math.sin(i/180*math.pi)
    b[i]=math.cos(i/180*math.pi)

plt.title("y = sin(x)") # заголовок
plt.xlabel("x") # ось абсцисс
plt.ylabel("y") # ось ординат
plt.grid(color='r', linestyle='--', linewidth=2) # включение отображение сетки

plt.plot(a.keys(),a.values(),linestyle = '---',color = (0, 0, 0, 1),linewidth = 12)
plt.plot(b.keys(),b.values())
plt.show()

```

В результате выполнению получим следующий график:

Рисунок 8 – Отображение двух графиков в одной рабочей области

Библиотека также позволяет отобразить два графика не накладывая их друг на друга.

```

import math
import matplotlib.pyplot as plt
a=dict()
b=dict()
for i in range(0, 360):
    a[i]=math.sin(i/180*math.pi)
    b[i]=math.cos(i/180*math.pi)

plt.figure(figsize=(9, 9))
plt.subplot(2, 1, 1)
plt.title("y = sin(x)") # заголовок
plt.xlabel("x") # ось абсцисс
plt.ylabel("y") # ось ординат
plt.grid(color='r', linestyle='--', linewidth=2) # включение отображение сетки
plt.plot(b.keys(),b.values())
plt.subplot(2, 1, 2)
plt.title("y = cos(x)") # заголовок
plt.xlabel("x") # ось абсцисс

```

```
plt.ylabel("y") # ось ординат
plt.grid(color='r', linestyle='--', linewidth=2) # включение отображение сетки
plt.plot(a.keys(),a.values(),linestyle = 'r',color = (0, 0, 0, 1),linewidth = 12)
plt.show()
```

Результатом выполнения кода будет следующий график:

Рисунок 9 – Отображение графиков в разных рабочих областях

Здесь мы воспользовались новыми функциями:

figure – функция для задания глобальных параметров отображения графиков. В нее, в качестве аргумента, мы передаем кортеж, определяющий размер общего поля (размер указывается в дюймах; при необходимости указания в сантиметрах, необходимо разделить нужные размеры на 2.54).

subplot – функция для задания местоположения поля с графиком. Существует несколько способов задания областей для вывода через функцию subplot() мы воспользовались следующим: первый аргумент – количество строк, второй – столбцов в формируемом поле, третий – индекс (номер поля, считаем сверху вниз, слева направо).

Тип графика

До этого момента мы работали только с линейными графиками. Функция **plot** позволяет задать тип графика: линейный либо точечный, при этом для точечного графика можно указать соответствующий маркер. Приведем пару примеров:

```
plt.plot(x, y, 'ro')
```

Рисунок 10 – Точечный график с круговым маркером

```
plt.plot(x, y, 'bx')
```

Рисунок 11 – Точечный график с маркером типа x

Первая буква задает цвет маркера ('b' – синий; 'g' – зеленый, 'r' – красный, 'c' – бирюзовый; 'm' – фиолетовый (пурпурный); 'y' – желтый; 'k' – черный; 'w' – белый), вторая буква задает тип маркера: '.' – точка, ',' – пиксель; 'o' – окружность; 'v' – треугольник, направленный вниз; '^' – треугольник, направленный вверх; '<' – треугольник, направленный влево; '>' – треугольник, направленный вправо; 's' – квадрат; 'p' – пятиугольник; '*' – звезда; 'h' – шестиугольник; '+' – плюс; 'x' – X-образный маркер; 'D' – ромб; '|' – вертикальная линия; '_' – горизонтальная линия. Размер маркера можно менять.

Столбчатые и круговые диаграммы

Помимо линейных графиков, библиотека Matplotlib позволяет также строить столбчатые и линейные диаграммы. Для их построения используются функции:

bar – для построения вертикальной диаграммы;

barh – для построения горизонтальной диаграммы;

pie – для построения круговых диаграмм.

Примеры:

```
import matplotlib.pyplot as plt

fruits=["apple","peach","orange","bannana","melon"]
counts = [34, 25, 43, 31, 17]

plt.bar(fruits, counts)
plt.title("Fruits!")
plt.xlabel("Fruit")
plt.ylabel("Count")
plt.show()
```

Рисунок 12 – Столбчатая диаграмма

```
import matplotlib.pyplot as plt
fruits=["apple","peach","orange","bannana","melon"]
counts = [34, 25, 43, 31, 17]
plt.barh(fruits, counts)
plt.title("Fruits!")
plt.xlabel("Fruit")
plt.ylabel("Count")
plt.show()
```

Рисунок 13 – Вертикальная столбчатая диаграмма

```
import matplotlib.pyplot as plt
fruits=["apple","peach","orange","bannana","melon"]
counts = [34, 25, 43, 31, 17]
plt.pie(counts,labels=fruits)
plt.title("Fruits!")
plt.show()
```

Рисунок 14 – Круговая диаграмма

Также как и линейные графики, диаграммы имеют множество настроек, которые можно узнать из официальной документации библиотеки.

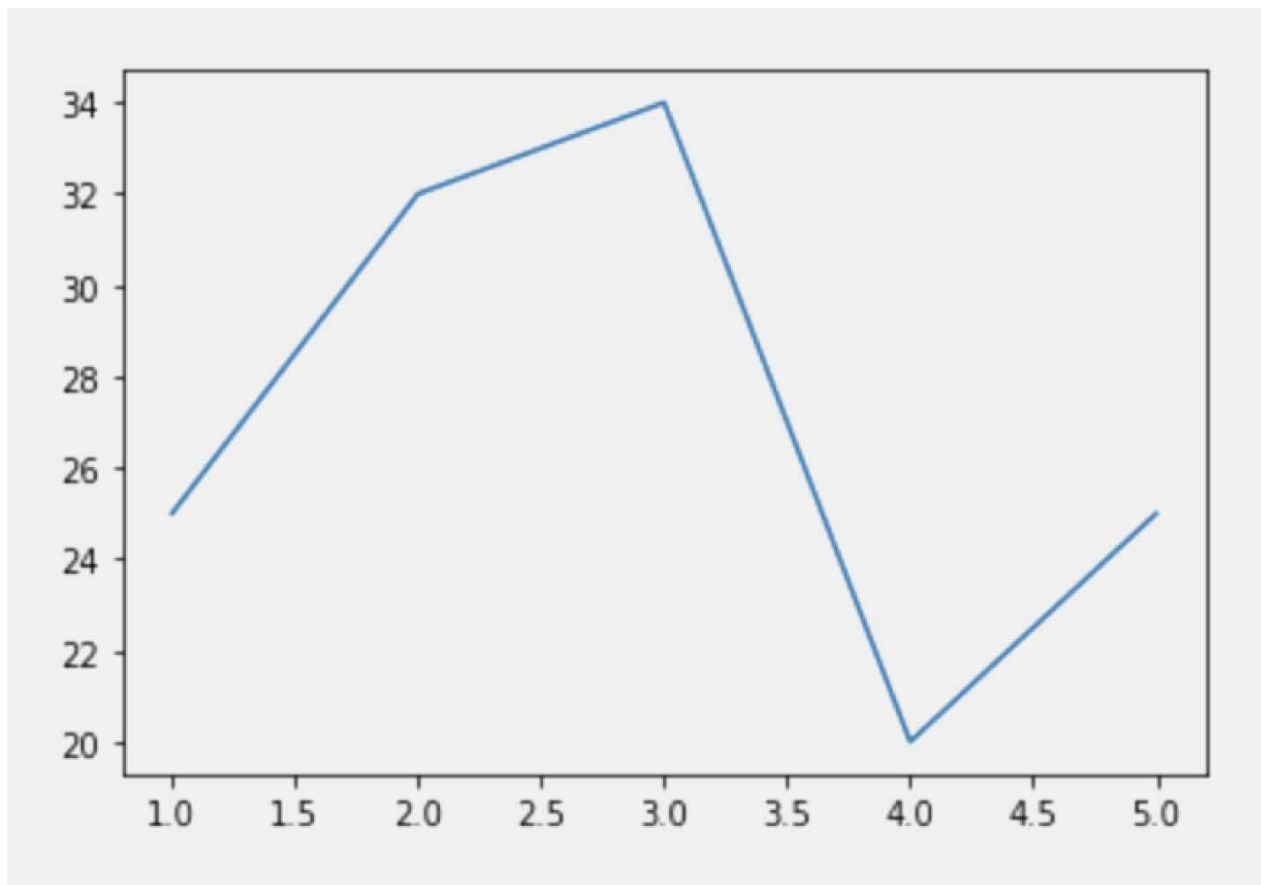
Примеры решения задания

Задание 1.

1.1. Построить простейший график по точкам:

```
x = [1, 2, 3, 4, 5]
y = [25, 32, 34, 20, 25]
plt.plot(x, y)
plt.show()
```

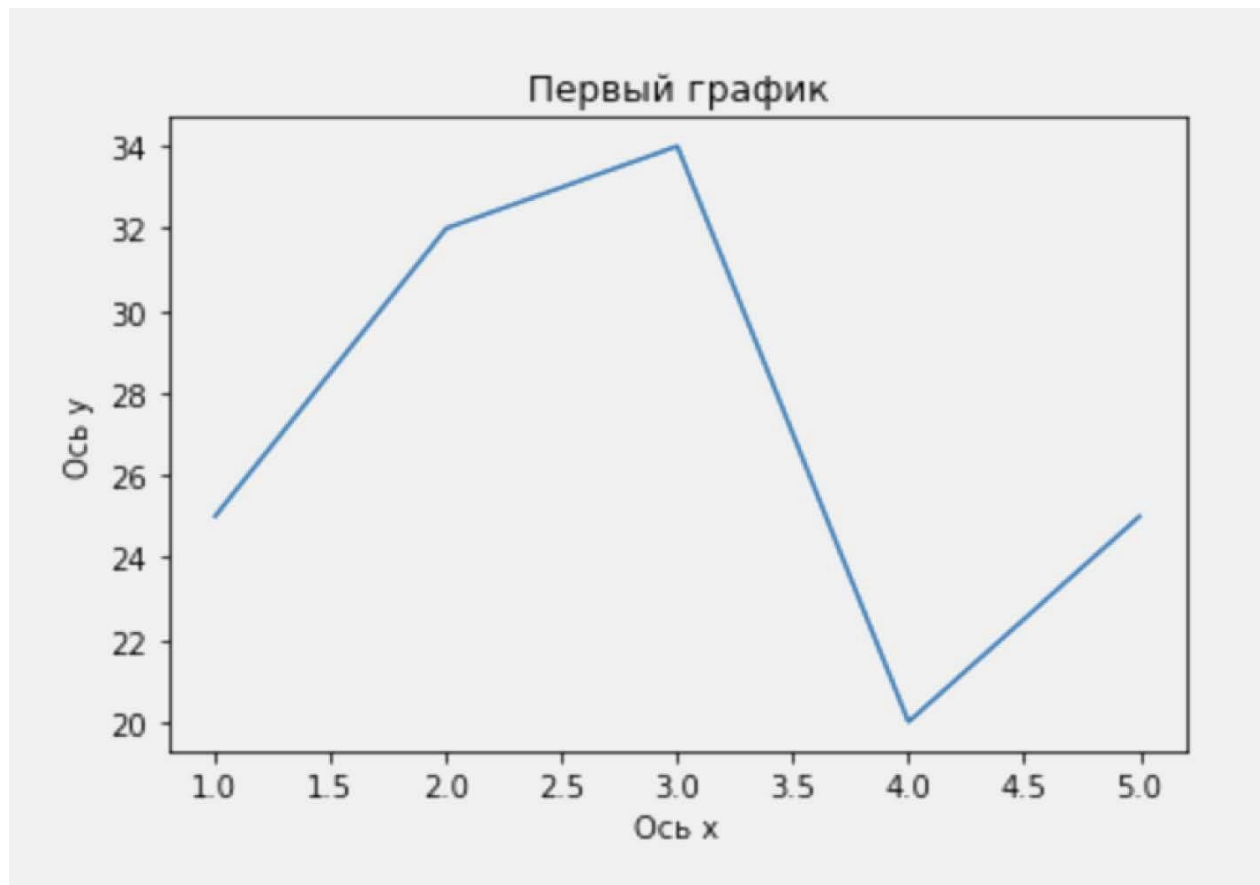
Результат выполнения:



1.2 Дополнить график заголовком и подписями осей.

```
plt.plot(x, y)
plt.xlabel('Ось x') #Подпись для оси x
plt.ylabel('Ось y') #Подпись для оси y
plt.title('Первый график') #Название
plt.show()
```

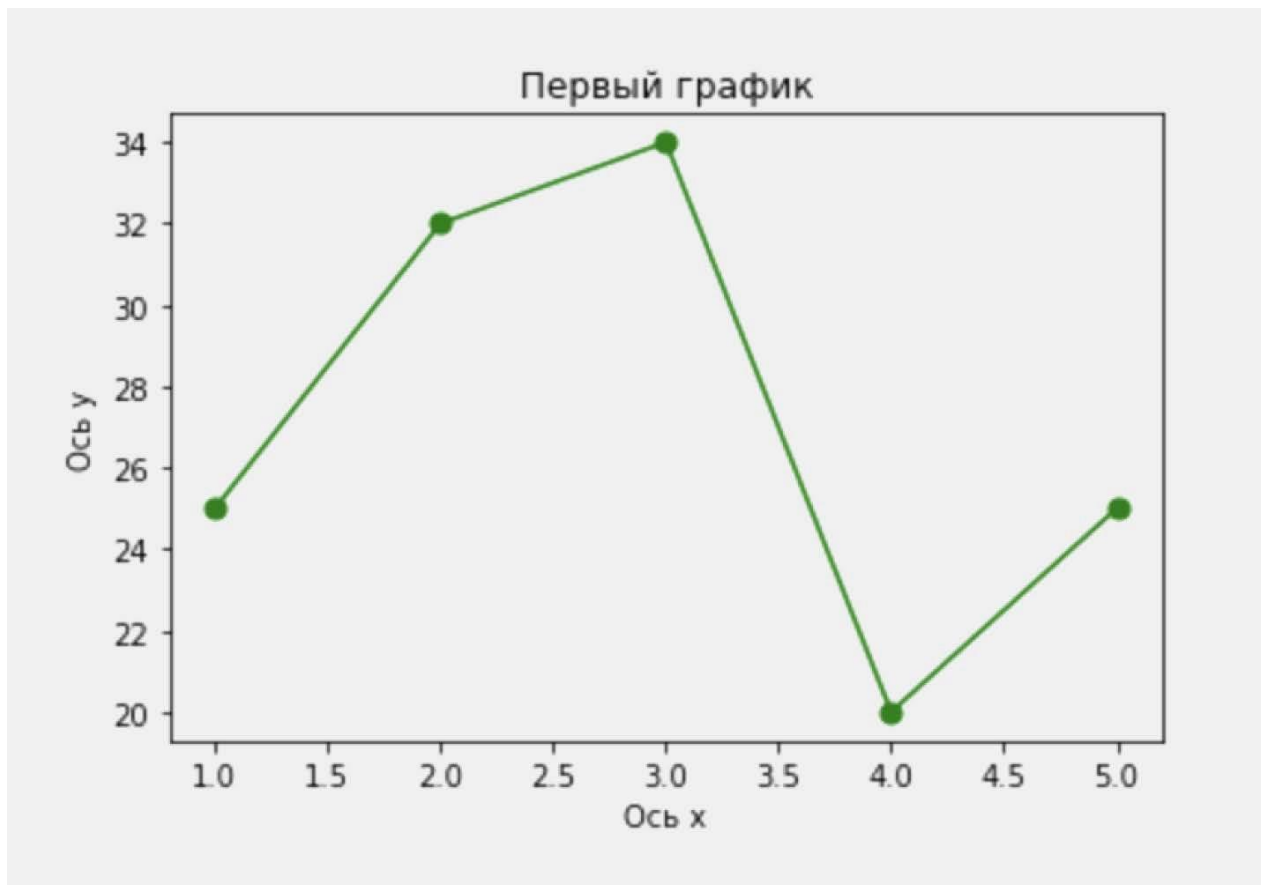
Результат



1.3 Изменить цвет линии и выделить точки, с заданными координатами.

```
plt.plot(x, y, color='green', marker='o', markersize=7)
```

Результат:



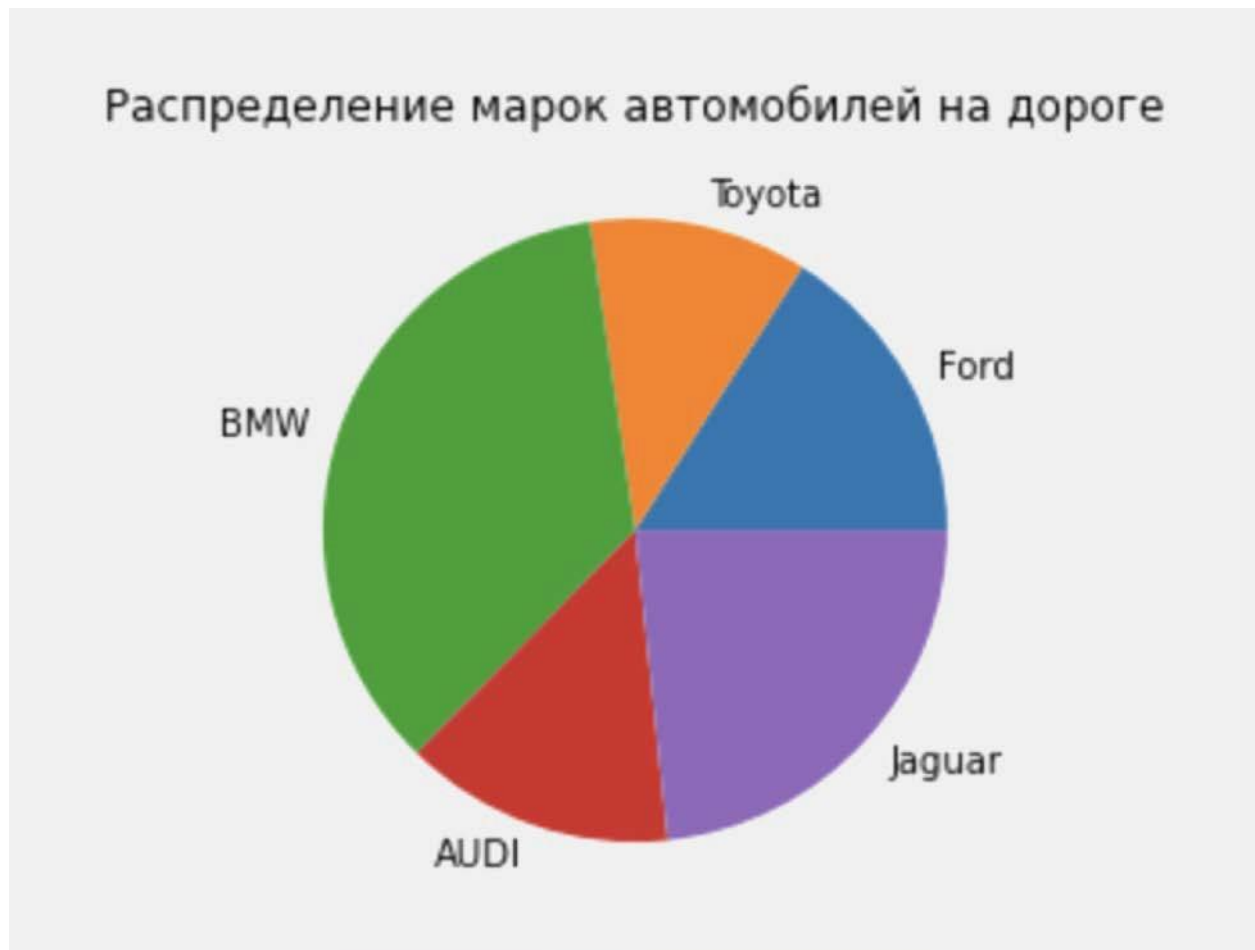
Задание 2.

2.1. Создать круговую диаграмму по следующим данным: vals = [24, 17, 53, 21, 35] и labels = ["Ford", "Toyota", "BMW", "Audi", "Jaguar"].

```
vals = [24, 17, 53, 21, 35]
labels = ["Ford", "Toyota", "BMW", "Audi", "Jaguar"]

plt.pie(vals, labels=labels)
plt.title("Распределение марок автомобилей на дороге")
plt.show()
```

Результат

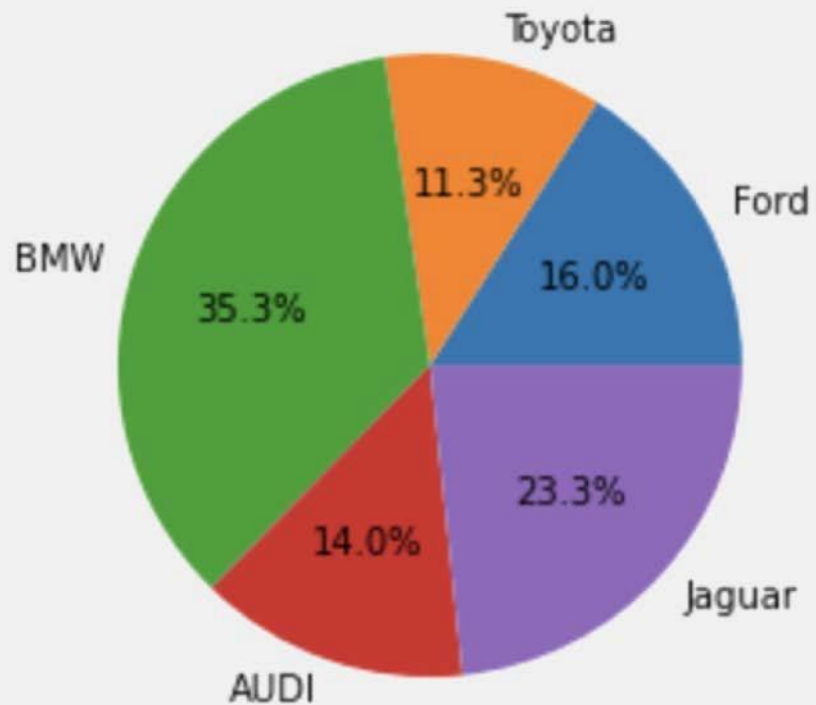


2.2. Дополнить круговую диаграмму значениями долей.

```
plt.pie(vals, labels=labels, autopct='%1.1f%%')  
plt.title("Распределение марок автомобилей на дороге")  
plt.show()
```

Результат

Распределение марок автомобилей на дороге



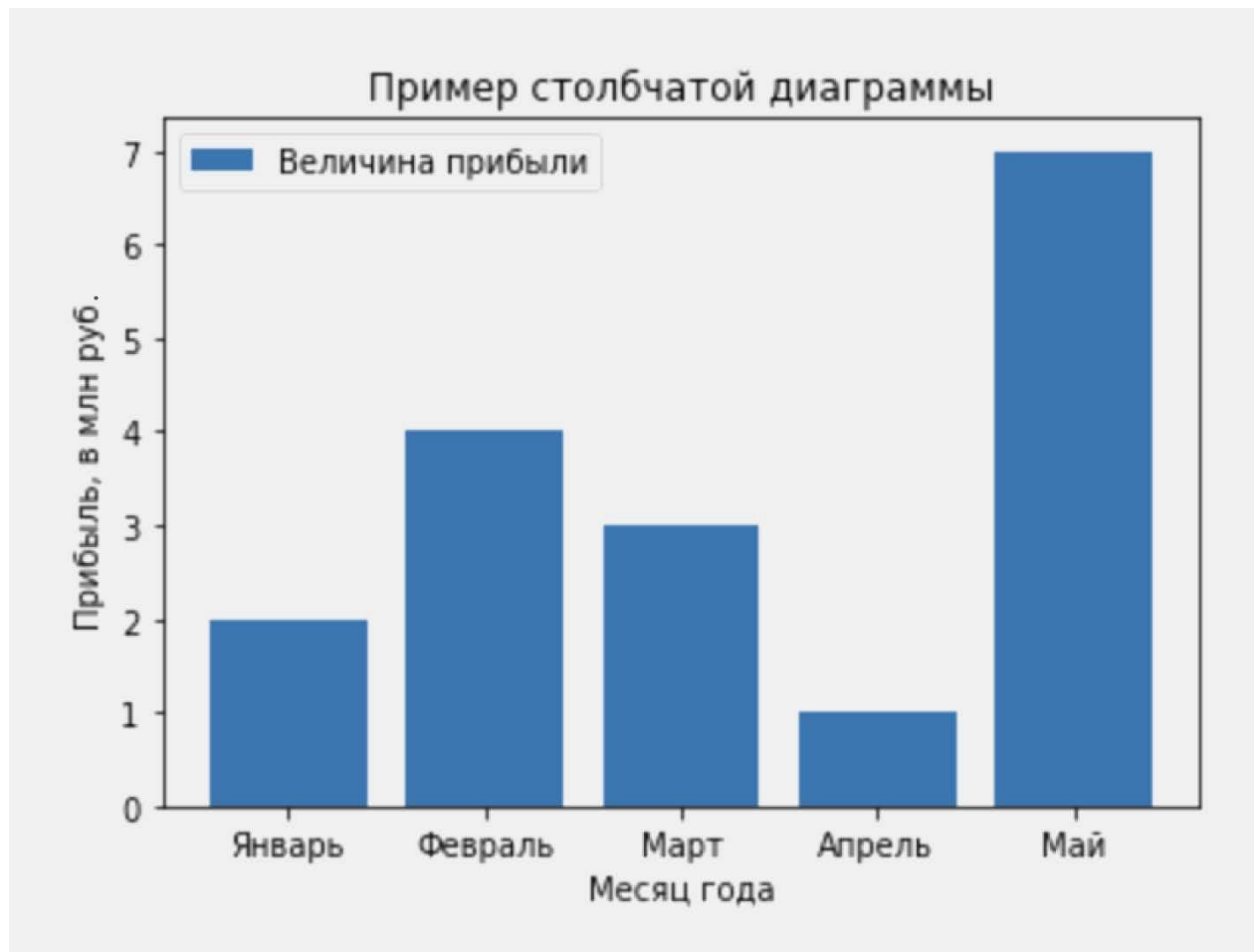
Задание 3.

2.1. Создать столбчатую диаграмму по следующим данным: $x =$ ['Январь', 'Февраль', 'Март', 'Апрель', 'Май'] и $y = [2, 4, 3, 1, 7]$

```
x = ['Январь', 'Февраль', 'Март', 'Апрель', 'Май']
y = [2, 4, 3, 1, 7]

plt.bar(x, y, label='Величина прибыли') #Параметр label позволяет задать
название величины для легенды
plt.xlabel('Месяц года')
plt.ylabel('Прибыль, в млн руб.')
plt.title('Пример столбчатой диаграммы')
plt.legend()
plt.show()
```

Результат



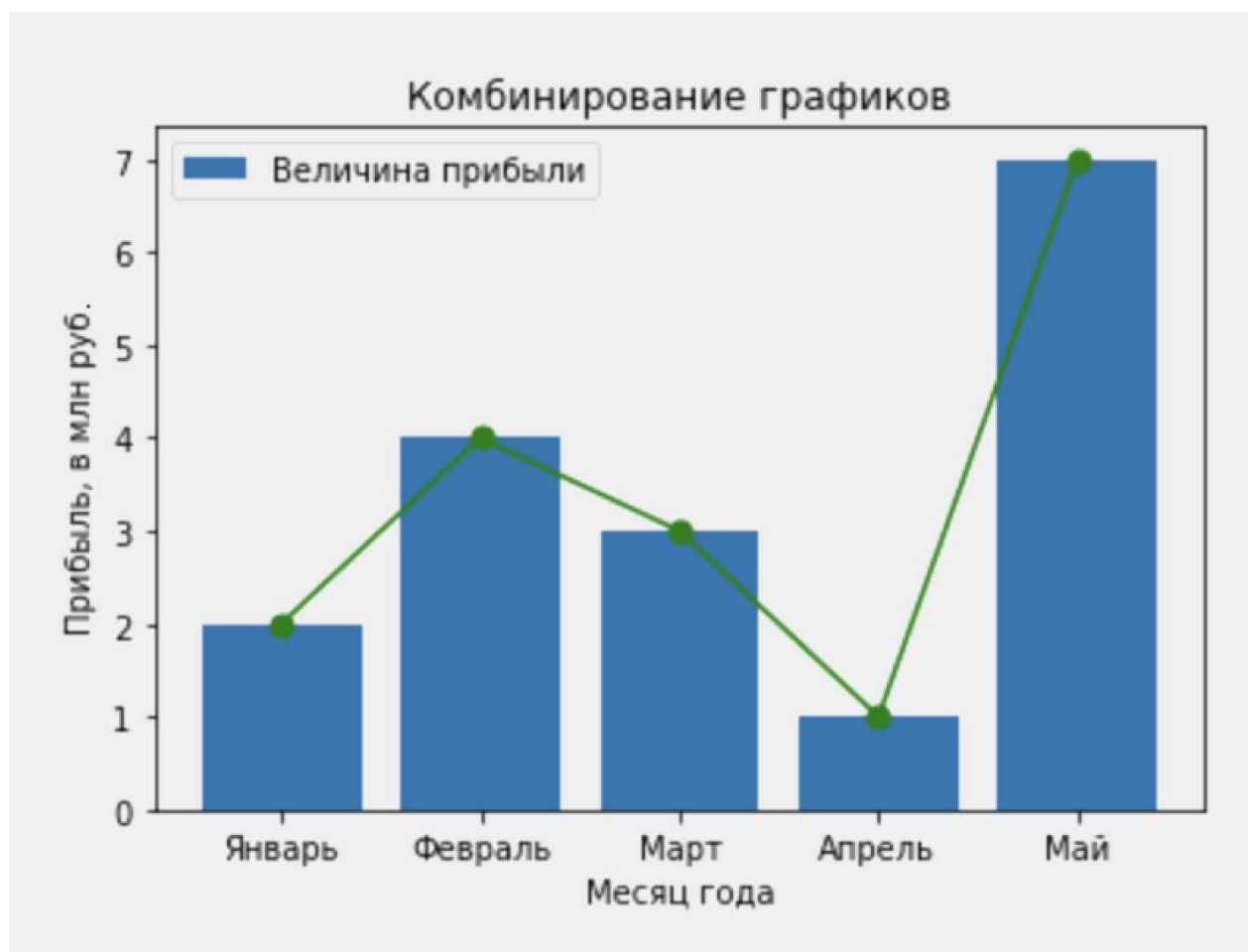
2.2. Дополнить столбчатую диаграмму линейным графиком.

```
x = ['Январь', 'Февраль', 'Март', 'Апрель', 'Май']
y = [2, 4, 3, 1, 7]

plt.bar(x, y, label='Величина прибыли') #Параметр label позволяет задать
название величины для легенды
plt.plot(x, y, color='green', marker='o', markersize=7)

plt.xlabel('Месяц года')
plt.ylabel('Прибыль, в млн руб.')
plt.title('Комбинирование графиков')
plt.legend()
plt.show()
```

Результат



Задание для самостоятельного выполнения

Задание 4.

Постройте график линейной функции $y=5x+8$ при $x=[-10,10]$.

Задание 5.

Постройте график синусоидальной функции при $x=[0,2\pi]$.

Задание 6.

Постройте график косинусоидальной функции при $x=[-\pi,\pi]$.

Задание 7.

Постройте график квадратичной функции $y=x^2$ при $x=[-10,10]$.

Задание 8.

Добавьте подписи осей: «ось x», «ось y».

Задание 9.

Добавьте заголовок «Задание 9».

Задание 10.

Измените цвет линии графика на розовый.

Задание 11.

Выделите координаты известных точек.

Задание 12.

Постройте круговую диаграмму по данным:

```
data = [33, 25, 20, 12, 10]
labels=('Group 1', 'Group 2', 'Group 3', 'Group 4', 'Group 5')
```

Задание 13.

Постройте круговую диаграмму по данным:

```
data = [70, 10, 10, 10]
labels=('В комментариях', 'В Ираке', 'В Сирии', 'В Афганистане')
```

Задание 14.

Постройте круговую диаграмму по данным:

```
data = [42, 36, 3, 7, 12]
labels=('Товар 1', 'Товар 2', 'Товар 3', 'Товар 4', 'Товар 5')
```

Задание 15.

Постройте круговую диаграмму по данным:

```
data = [35, 26, 24, 9, 6]
labels=('Боевик', 'Love story', 'Приключения', 'Исторический', 'Комедия')
```

Задание 16.

Постройте столбчатую диаграмму по данным:

```
objects = ('A', 'B', 'C', 'D', 'E', 'F')
performance = [10, 8, 6, 4, 2, 1]
```

Задание 17.

Постройте столбчатую диаграмму по данным:

```
objects = ('A', 'B', 'C', 'D', 'E', 'F')
performance = [10, 8, 6, 4, 2, 1]
```

Задание 18.

Постройте столбчатую диаграмму по данным:

```
objects = ('январь', 'март', 'июнь', 'сентябрь', 'декабрь')
performance = [378, 356, 210, 568, 456]
```

Задание 19.

Постройте столбчатую диаграмму по данным:

```
objects = ('15-20', '21-25', '26-30', '31-35', '36-40', '41-45', '46-50',  
'50+')  
performance = [7, 11, 24 29 26 19 8 4]
```

Контрольные вопросы

1. Для чего нужна библиотека Matplotlib?
2. Расскажите об основных терминах и понятиях библиотеки Matplotlib.
3. Какие основные типы графиков и диаграмм можно построить с помощью библиотеки Matplotlib?
4. Какая функция используется для построения линейных графиков?
Какие у нее есть параметры?
5. Какая функция используется для построения точечных графиков?
Какие у нее есть параметры?
6. Какая функция используется для построения столбчатых диаграмм?
7. Какая функция используется для построения круговых диаграмм?
8. Как отобразить два и более графиков на одном листе?

Практическое занятие №7

Тема: Объектно-ориентированное программирование, визуализация данных.

Цель работы: Познакомиться с объектно-ориентированным программированием, понятием класс, атрибутами и методами класса, способами его объявления, создания элементов класса.

Теоретическая часть

До сих пор мы разрабатывали программы, используя методологию структурного программирования. Структурное программирование – методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков.

Методология структурного программирования предложена в 1970-х годах и была прорывной для того времени, так как позволила формализовать процесс разработки программ, выделять часто используемые блоки кода и многократно их использовать.

Однако с ростом сложности программ начали проявляться и недостатки структурного программирования. Основной недостаток структурного программирования заключается в том, что данные (переменные) и подпрограммы (функции) формально не связаны, что плохо подходит для описания реальной картины мира.

Объектно-ориентированное программирование (ООП) – подход к программированию, при котором основными концепциями являются понятия объектов и классов.

Класс – это определяемый разработчиком тип данных.

Тип данных характеризуется:

- способом хранения и представления этих данных;
- назначением этих данных;
- набором действий, которые над этими данными могут производиться.

Чтобы понять, для чего же в действительности нужны классы, проведём аналогию с каким-нибудь объектом из повседневной жизни, например, с велосипедом. Велосипед — это объект, который был построен согласно чертежам.

Так вот, эти самые чертежи играют роль классов в ООП.

Таким образом классы — это некоторые описания, схемы, чертежи по которым создаются объекты. Для создания объекта в ООП необходимо сначала составить чертежи, то есть классы.

Классы имеют свои функции, которые называются методами класса и свои параметры, которые называются полями, или свойствами класса.

Создание класса в Python начинается с инструкции `class`. Вот так будет выглядеть минимальный класс:

```
1. class C:  
2.     pass
```

Класс состоит из объявления (инструкция `class`), имени класса (нашем случае это имя `C`) и тела класса, которое содержит атрибуты и методы (в нашем минимальном классе есть только одна инструкция `pass`).

Для того чтобы создать объект класса необходимо воспользоваться следующим синтаксисом:

```
имя_объекта = имя_класса()
```

Как уже было сказано, класс может содержать атрибуты и методы. Добавление методов происходит аналогично созданию функций. Но есть одна особенность языка Python, которую нужно иметь в виду. Первым аргументом метода должно быть ключевое слово «`self`». `self` — это ссылка на текущий экземпляр класса. Названия методов пользователь придумывает сам. Но есть некоторые заранее зарезервированные имена методов, которые выполняют особую функцию. Один из таких методов — метод «`__init__`». Он вызывается каждый раз при создании нового объекта. В этом методе должны быть описаны все атрибуты класса.

Примеры решения задания

Задача:

Создайте класс «узел сети», который имеет следующие свойства:

- номер узла;
- тип узла (базисный или нагрузочный);
- номинальное напряжение узла;
- расчетное напряжение.

Решение:

Давайте подумаем, какие типы данных лучше всего подходят для моделирования этих свойств в программе.

Номер узла – это целое число, поэтому его проще всего моделировать с помощью переменной типа `int`.

Тип узла может принимать два значения – базисный и нагрузочный. Поэтому его лучше моделировать с помощью переменной типа `bool`.

Номинальное напряжение представляет собой десятичное число, поэтому целесообразно моделировать с помощью переменной `float`.

Расчетное напряжение представляет собой комплексное число. Для его моделирования можно использовать две переменные типа `float`. Первое число будет моделировать модуль напряжения, второе – фазовый угол. Однако в данном случае целесообразно использовать комплексное число, поскольку это позволит использовать уравнения расчета режимов в комплексной форме.

С учетом сказанного, описание класса узла будет выглядеть следующим образом:

```
class NODE:                                # Класс "Узел"
    def __init__(self):                    # Конструктор класса
        self.ny = 0                        # Номер узла
        self.type = False                 # Тип узла
        self.unom = 0.0                   # Номинальное напряжение
        self.V = 0+0j                     # Расчетное напряжение
```

Задача:

Создайте класс, описывающий схему сети, имеющий следующие свойства:

- таблица узлов сети;
- и следующие методы:
- загрузить таблицу узлов из файла;
- сохранить таблицу узлов в файл.

Решение

Для хранения таблицы узлов будем использовать список.

```
class SCHEME:                                # Класс "Схема"
    def __init__(self):                       # Конструктор класса
        self.nodes=[]                        # Таблица узлов
```

Чтобы реализовать загрузку таблицы узлов из файла для начала необходимо определиться с форматом входного файла. Формат файла – это совокупность правил записи исходной информации.

Наиболее простым и удобным для создания, просмотра, редактирования и использования являются файлы CSV – текстовые файлы с разделителями. Удобные они потому, что для их создания и редактирования можно использовать программу Excel. Предположим, что входной файл имеет следующий формат:

	A	B	C	D
1	№	Тип	Unom	
2		1 База	110	
3		2 Нагр	110	
4		3 Нагр	110	
5				
6				

После сохранения в формате «csv», его можно открыть в программе «Блокнот». Файл будет иметь следующий вид:

Файл	Правка	Формат	Вид	Справка
№;Тип;Unom				
1;База;110				
2;Нагр;110				
3;Нагр;110				

Для чтения таких файлов можно использовать функцию DictReader модуля CSV. Создадим в классе «SCHEME» метод load, в который будем передавать путь к файлу с таблицей узлов в предложенном формате.

```
class SCHEME:
    # Класс "Схема"
    def __init__(self):
        # Конструктор класса
        self.nodes=[]
        # Таблица узлов

    def load(self, fname):
        # Функция загрузки таблицы узлов из файла
        file=open(fname)
        # Открываем файл для чтения
        for line in csv.DictReader(file, delimiter=";"):
            # Читаем файл построчно в словарь
            n = NODE()
            # Создаем новый узел
            # Заполняем свойства узла данными, прочитанными из файла
            n.ny = int(line['№'])
            if line['Тип']=='База':
                n.type=True
            n.unom = float(line['Unom'])
            # Добавляем узел в таблицу узлов
            self.nodes.append(n)
        file.close()
```

Для сохранения в файл создадим метод save, в который будем передавать путь к файлу для сохранения. При сохранении в файл будем использовать «любимую» функцию print:

```
class SCHEME:
    # Класс "Схема"
    def __init__(self):
        # Конструктор класса
        self.nodes=[]
        # Таблица узлов

    def load(self, fname):
        # Функция загрузки таблицы узлов из файла
        file=open(fname)
        # Открываем файл для чтения
        for line in csv.DictReader(file, delimiter=";"):
            # Читаем файл построчно в словарь
            n = NODE()
            # Создаем новый узел
            # Заполняем свойства узла данными, прочитанными из файла
            n.ny = int(line['№'])
            if line['Тип']=='База':
                n.type=True
            n.unom = float(line['Unom'])
            # Добавляем узел в таблицу узлов
            self.nodes.append(n)
        file.close()

    def save(self, fname):
        # Функция сохранения таблицы узлов в файл
        f=open(fname, "w")
        # Открываем файл для записи
        print("Узлы", file=f)
        # Выводим в файл название таблицы
        for n in self.nodes:
            # Обходим все узлы
            print(n.ny, end='\t', file=f)
            # Сохраняем в файл параметры узла
            print(n.type, end='\t', file=f)
            # Сохраняем в файл параметры узла
            print(n.unom, end='\t', file=f)
            # Сохраняем в файл параметры узла
            print(n.V, file=f)
            # Сохраняем в файл параметры узла
        f.close()
        # Закрываем файл
```

После того, как класс схемы сети описан, его можно использовать непосредственно в нашей программе:

```
s=SCHEME()  
s.load("C:\\input.csv")  
s.save("E:\\output.txt")
```

В первой строке создается объект типа «Схема».

Во второй строке вызывается метод load, в который передается имя входного файла, содержащего информацию о таблице узлов.

В третьей строке выполняется сохранение таблицы узлов в соответствующий файл.

Задание для самостоятельного выполнения

1. Создайте класс, описывающий узел сети, который должен иметь следующие свойства:

- номер узла;
- имя узла;
- тип узла (базисный или нагрузочный);
- номинальное напряжение узла;
- активная мощность нагрузки;
- реактивная мощность нагрузки;
- расчетное напряжение;
- дополнительные свойства согласно вашему номеру варианта

(таблица 1).

2. Создайте класс, описывающий ветвь сети, который должен содержать следующие свойства:

- номер узла начала ветви;
- номер узла конца ветви;
- активное сопротивление линии;
- реактивное сопротивление линии;
- активная проводимость;
- реактивная проводимость;

– коэффициент трансформации;
 – дополнительные свойства согласно вашему номеру варианта (таблица 1).

3. Создайте класс, описывающий схему сети, который будет содержать следующие свойства:

- список узлов сети;
- список ветвей сети;

и следующие методы:

- загрузить схему сети из файла;
- вывести таблицу узлов и таблицу ветвей на экран;
- сохранить схему сети в файл.

Формат файла должен соответствовать вашему номеру варианта (таблица 1).

4. Нарисуйте диаграмму созданных классов.

Таблица 1 – Задания по вариантам

№ варианта	Дополнительные свойства классов узлов и ветвей	
	Таблица узлов	Таблица ветвей
1	– минимально допустимое напряжение – максимально допустимое напряжение	– длина линии
2	– номер района	– максимально допустимый ток
3	– минимальная генерация реактивной мощности – максимальная генерация реактивной мощности	– марка провода
4	– минимально допустимое напряжение – максимально допустимое напряжение	– мощность трансформатора
5	– номер района	– площадь сечения провода
6	– минимальная генерация реактивной мощности – максимальная генерация реактивной мощности	– марка трансформатора
7	– минимально допустимое напряжение – максимально допустимое напряжение	– положение отпайки трансформатора
8	– номер района	– номер района
9	– минимальная генерация реактивной мощности – максимальная генерация реактивной мощности	– температура
10	– минимально допустимое напряжение – максимально допустимое напряжение	– длина линии
11	– номер района	– максимально допустимый ток
12	– минимальная генерация реактивной мощности – максимальная генерация реактивной мощности	– марка провода
13	– минимально допустимое напряжение – максимально допустимое напряжение	– мощность трансформатора
14	– номер района	– площадь сечения провода
15	– минимальная генерация реактивной мощности – максимальная генерация реактивной мощности	– марка трансформатора

Контрольные вопросы

1. Что такое объектно-ориентированное программирование?
2. Что такое класс? Какие параметры есть у класса?
3. Как объявлять класс?
4. Как создать объект класса?
5. Как использовать атрибуты и методы класса?

Практическое занятие №8

Тема: Объектно-ориентированное программирование, визуализация данных.

Цель работы: Познакомиться с примером решения профессионально-ориентированных задач с помощью языка Python. Разработка алгоритма расчета установившегося режима методом Зейделя.

Теоретическая часть

В электроэнергетических системах модель управляемого объекта в самом упрощенном варианте можно представить в виде множества узлов и ветвей. Ветви разделяются на продольные и поперечные. Каждый узел связан с хотя бы одним другим узлом схемы сети посредством продольных ветвей. Узел описывается мощностью, проводимостью поперечной ветви и напряжением (рисунок 1).

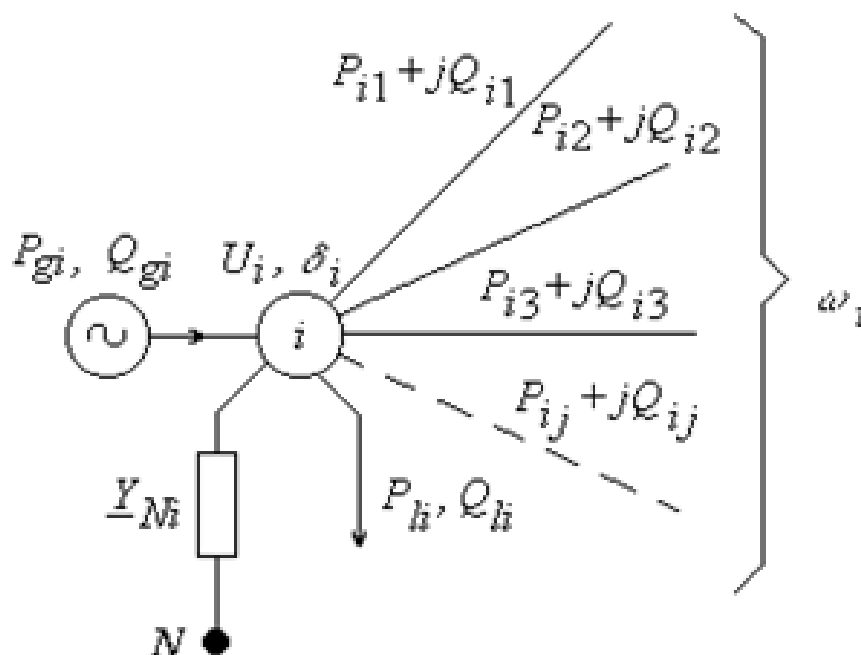


Рисунок 1 – Модель узла схемы сети

Для узла i :

P_{gi} – активная мощность генерации;

Q_{gi} – реактивная мощность генерации;

P_{li} – активная мощность нагрузки;

Q_{li} – реактивная мощность нагрузки;

$\underline{Y}_{Ni}$ – проводимость поперечной ветви;

U_i, δ_i – модуль и фаза напряжения;

ω_i – множество номеров узлов, смежных i -у.

Мощность нагрузки задается постоянной величиной или статическими характеристиками.

Продольные ветви являются элементами схем замещения ЛЭП или трансформатора.

Расчет установившихся режимов ЭС в классическом виде заключается в определении напряжений в узлах сети, используя которые, затем находят потокораспределение и потери мощности. Математически эта задача формулируется как решение системы нелинейных алгебраических уравнений, описывающих режим. В основе такого описания состояния ЭС лежат законы Кирхгофа и Ома, устанавливающие связи между токами, напряжениями и параметрами сети. Непосредственное применение уравнений Кирхгофа неэффективно и в алгоритмах для ЭВМ не используется. Для построения соответствующих алгоритмов электрического расчета наиболее эффективными и удобными для реализации на ЭВМ являются уравнения узловых напряжений, связывающие напряжения в узлах ЭС и мощности (токи), подводимые к этим узлам, через параметры схемы. Уравнения узловых напряжений (УУН) следуют из первого закона Кирхгофа в результате представления по закону Ома токов во всех ветвях через узловые напряжения и проводимости ветвей.

Методы, используемые в современных программно-вычислительных комплексах для расчета установившихся режимов, могут быть разделены на две основные группы:

1) методы, предусматривающие на каждой итерации последовательное решение уравнений, относящихся к каждому узлу, друг за другом, так

называемые поузловые методы;

2) методы, предусматривающие на каждой итерации одновременное решение системы уравнений сразу для всех узлов.

К первой группе методов относится метод Зейделя. Коэффициенты уравнений и переменные при этом могут записываться либо в комплексной форме, либо в прямоугольной системе координат.

Таким образом, метод Зейделя основан на решении в данной итерации последовательно для одного узла за другим уравнений следующего вида:

$$U_i^{(k)} = \frac{1}{Y_{ii}} \left(\frac{S_i}{U_i^{(k-1)}} - \sum_{j=1, j \in G_i}^n Y_{ij} U_j^{(k-1)} \right), \quad (1)$$

используя значения напряжений, найденные из предшествующих уравнений системы на этой же итерации.

Здесь k – номер итерации;

Y_{ij} – взаимная проводимость между узлами i и j ;

Y_{ii} – собственная узловая проводимость узла i ;

G_i – множество узлов индекса j , непосредственно связанных с i -м узлом;

S – полная мощность нагрузки ($P+jQ$).

В алгоритмическом отношении этот метод весьма прост, так как определение напряжения в каждом из узлов на итерации производится по единственной формуле (1). Уравнение (1) применяется для всех узлов, кроме базисных (узлов, в которых заранее известно значение напряжения). Исходя из теории расчетов режимов, в схеме должен быть как минимум один базисный узел. Для базисных узлов формула (1) применяется в модифицированном виде: необходимо его преобразовать, чтобы выразить из него переменную S . Широкий опыт расчетов режимов этим методом в прошлом показал, что итерационный процесс решения носит монотонный характер и сходится, несмотря на возможность ускорения сходимости, весьма медленно.

Примеры решения задания

Примеры решения задания

Задание

Составьте алгоритм расчета режима методом Зейделя.

Решение

Алгоритм решения будет включать в себя следующие этапы:

1. Загрузить схему сети из файла.
2. Для каждого узла задать начальные приближения расчетного напряжения, равное номинальному напряжению узла.

3. Для каждой ветви посчитать проводимость Y_{ij} по формуле:

$$Y_{ij} = \frac{1}{R_{ij} + jX_{ij}}$$

4. Для каждого узла посчитать собственную узловую проводимость по формуле:

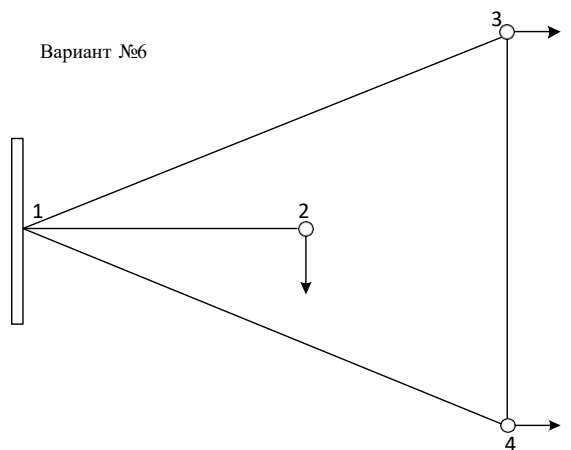
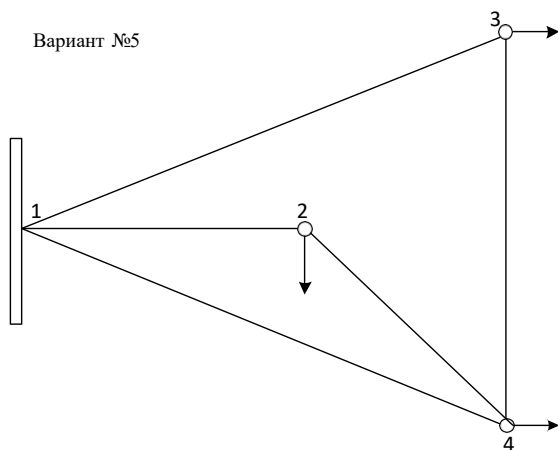
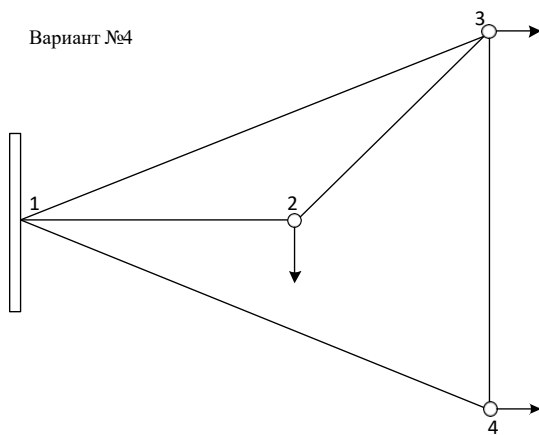
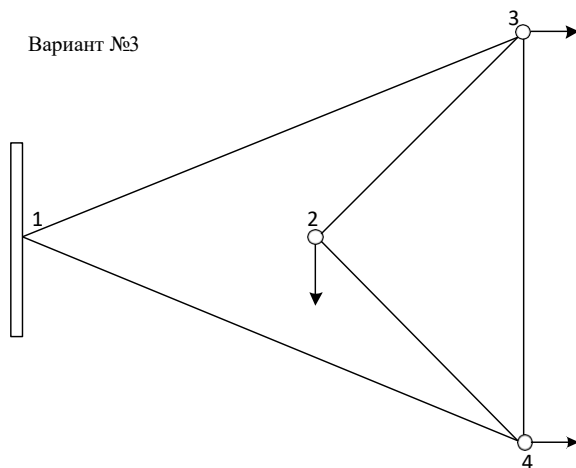
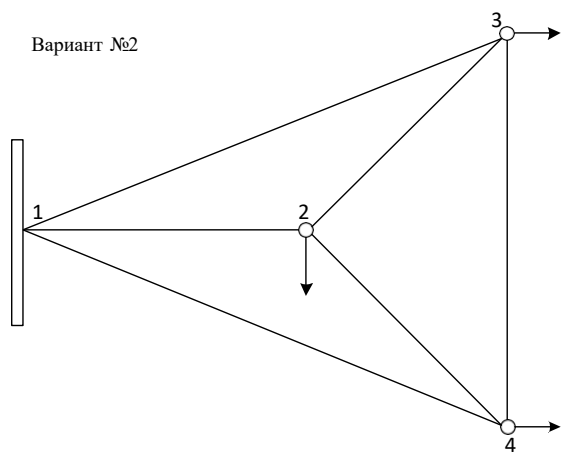
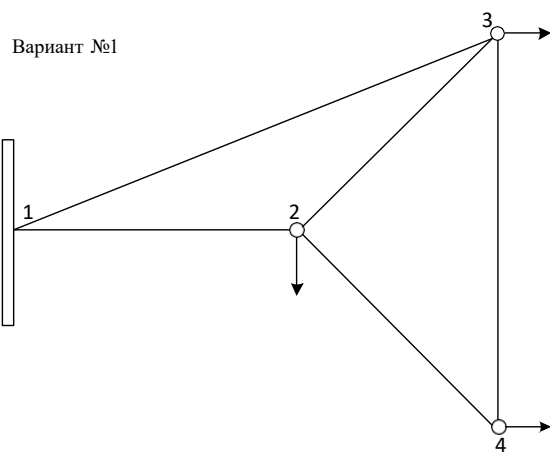
$$Y_{ii} = -\sum_{j=1}^n Y_{ij}$$

5. Для каждого узла посчитать формулу (1)

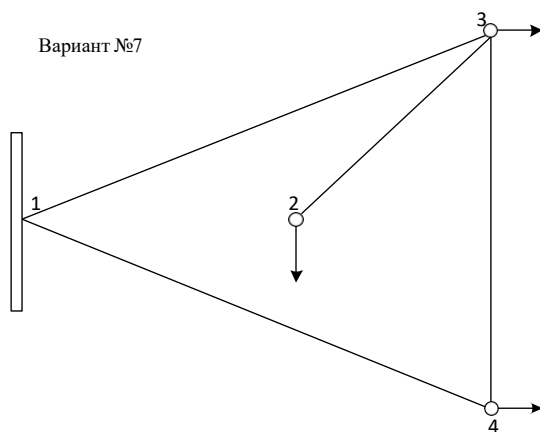
6. Повторить операцию 5 100 раз.

Задание для самостоятельного выполнения

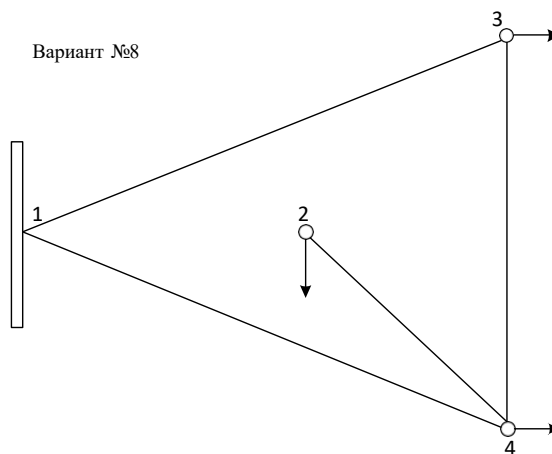
1. Составьте алгоритм расчета режима методом Зейделя. Нарисуйте его блок-схему.
2. Подготовьте исходный файл со схемой сети, согласно вашему номеру варианта. Протестируйте работоспособность созданных в предыдущей работе классов и методов для загрузки схемы сети.



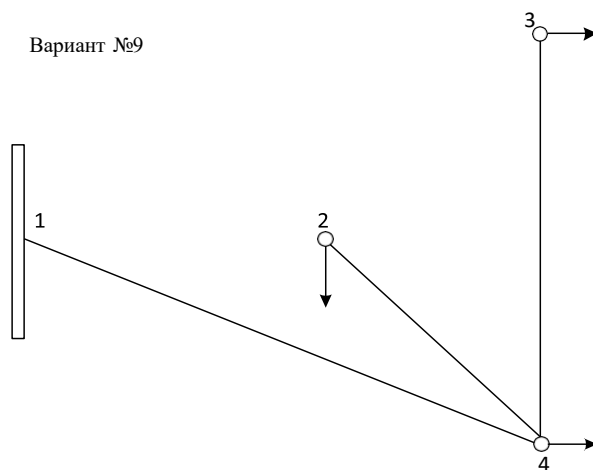
Вариант №7



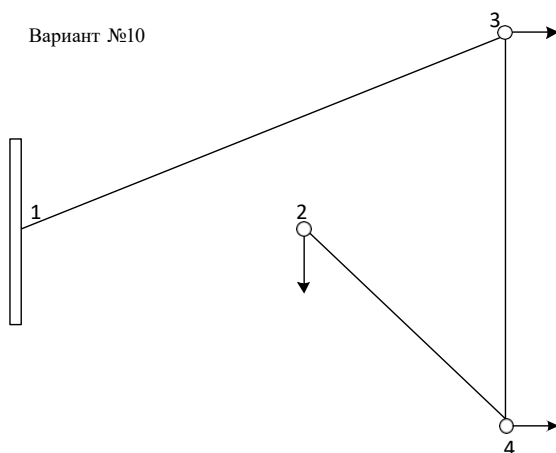
Вариант №8



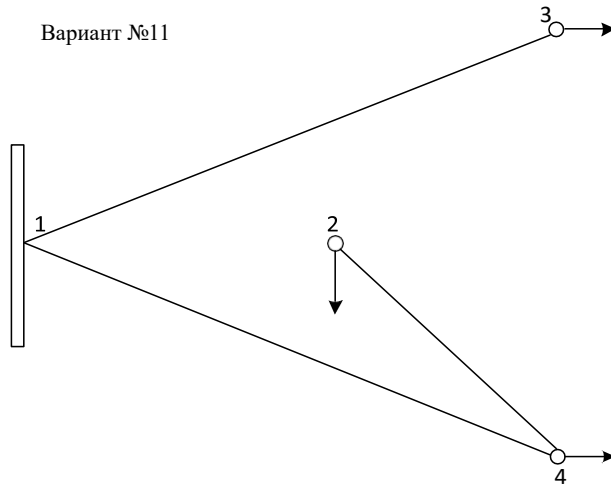
Вариант №9



Вариант №10



Вариант №11



Вариант №12

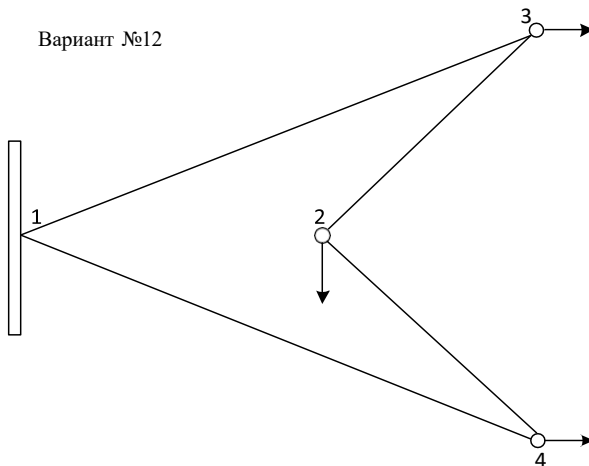


Таблица 1 – Параметры схемы сети

Вариант	Длина каждой ЛЭП, км	U1, кВ	P2, МВт	Q2, Мвар	P3, МВт	Q3, Мвар	P4, МВт	Q4, Мвар
1	15	113,00	5,00	2,56	15,00	7,68	17,00	6,72
2	10	115,00	2,00	0,73	14,00	7,17	16,00	9,49
3	20	112,00	2,00	0,79	13,00	4,27	11,00	4,69
4	18	114,00	10,00	5,93	12,00	6,15	14,00	6,38
5	8	114,00	7,00	3,78	10,00	5,67	15,00	9,30
6	30	110,00	6,00	1,97	14,00	7,93	12,00	7,44
7	10	112,00	8,00	4,53	12,00	3,94	18,00	9,72
8	12	115,00	1,00	0,59	11,00	5,01	16,00	5,26
9	14	110,00	9,00	3,56	13,00	6,66	19,00	7,51
10	16	114,00	10,00	3,95	5,00	2,83	19,00	7,51
11	17	111,00	6,00	2,91	14,00	5,96	19,00	10,77
12	18	115,00	5,00	2,13	10,00	5,40	19,00	7,51

Сопротивление провода: $R=0.198 \text{ Ом/км}$, $X=0.42 \text{ Ом/км}$

Контрольные вопросы

1. Опишите модель узла сети.
2. В чем заключается задач расчета установившихся режимов энергосистемы в классическом виде?
3. Какие уравнения используются при расчете режимов?
4. Опишите основные этапы метода Зейделя.

Практическое занятие №9

Тема: Основные сведения о базах данных, обработка данных.

Библиотека NumPy.

Цель работы: Познакомиться с Библиотекой NumPy, процедурами подключения сторонних библиотек, многомерными массивами однородных данных, матрицами, универсальными функциями библиотеки.

Теоретическая часть

Основной объект в NumPy — многомерный массив. За тип массива здесь отвечает объект `array`.

Массив можно сделать из обычного списка:

```
import numpy as np
np.array([1, 4, 10, 34])
```

Массив можно сделать из диапазона:

```
np.array(range(10))
#array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Массив можно сделать из итератора с помощью функции `fromiter()`:

```
np.fromiter(map(int, ["1", "2", "3", "4"]), dtype=np.int8)
#array([1, 2, 3, 4], dtype=int8)
```

В NumPy элементы одного массива должны быть однородны (одного типа). Это самое важное идеологическое отличие массивов от списков, в которых можно хранить объекты разной природы

Указание типов и работа с ними нужны, поскольку языки, на которых написана эта библиотека, строго типизованы. Вдобавок это увеличивает скорость обработки данных.

Размерность массива можно в любой момент изменить операцией `reshape()`. Узнать размерность можно атрибутом `shape`.

```
a = np.arange(100)
```

```
a.shape
```

```
a.reshape(10, 10)
```

Самое главное — при использовании функции `reshape()` произведение ее параметров должно быть равно количеству элементов в массиве.

Примеры решения задания

Задание: создать матрицу A: $a_{i,j}=i+j*2$, размер 2x3

Создать матрицу B: $b_{i,j}=0..5$, размер 3x2

Рассчитать $C=A \times B$

Решение

```
import numpy as np  
  
a = np.array(range(6)).reshape(2, 3)  
  
for i in range(2):  
    for j in range(3):  
        a[i][j] = i + (j * 2)  
  
b = np.array(range(6)).reshape(3, 2)  
  
print("a=", a)  
print("b=", b)  
print("c=", a.dot(b))
```

Результат

a= [[0 2 4]
[1 3 5]]

b= [[0 1]
[2 3]
[4 5]]

c= [[20 26]
[26 35]]

Задание для самостоятельного выполнения

1. Выполнить действия (по вариантам)

1. Создать матрицу A: $a_{i,j}=5*i+2*j$, размер 2x3 Создать матрицу B: $b_{i,j}=0..5$, размер 2x3 Рассчитать $C=A+B$
2. Создать матрицу A: $a_{i,j}=(i+1)+2*j$, размер 2x3 Создать матрицу B: $b_{i,j}=0..5$, размер 2x3 Рассчитать $C=A-B$
3. Создать матрицу A: $a_{i,j}=i+j+2$, размер 2x3 Создать матрицу B: $b_{i,j}=0..5$, размер 3x2 Рассчитать $C=A+B^T$
4. Создать матрицу A: $a_{i,j}=5*(i+j)$, размер 2x3 Создать матрицу B: $b_{i,j}=0..5$, размер 2x3 Рассчитать $C=A+B$
5. Создать матрицу A: $a_{i,j}=3*i+3*j$, размер 2x3 Создать матрицу B: $b_{i,j}=0..5$, размер 3x2 Рассчитать $C=A*B$
6. Создать матрицу A: $a_{i,j}=2*(i-j)$, размер 2x3 Создать матрицу B: $b_{i,j}=0..5$, размер 2x3 Рассчитать $C=A-B$

7. Создать матрицу A: $a_{i,j}=2*i+3*j$, размер 2x3 Создать матрицу B: $b_{i,j}=0..5$, размер 2x3 Рассчитать $C=A+B$
8. Создать матрицу A: $a_{i,j}=3*i+j$, размер 2x3 Создать матрицу B: $b_{i,j}=0..5$, размер 2x3 Рассчитать $C=A-B$
9. Создать матрицу A: $a_{i,j}=4i-j$, размер 2x3 Создать матрицу B: $b_{i,j}=0..5$, размер 3x2 Рассчитать $C=A*B$
10. Создать матрицу A: $a_{i,j}=6*i-6*j+1$, размер 2x3 Создать матрицу B: $b_{i,j}=0..5$, размер 2x3 Рассчитать $C=A+B$

2. Полученную в задании 1 матрицу C:

- а) транспонировать;
- б) повернуть влево на 90^0 ;
- в) заполнить единицами

3. Решить СЛАУ (по вариантам)

1. $2x_1+3x_2+4x_3=12$ $x_1+4x_2=8$ $2x_2+5x_3+x_3=4$
2. $3x_1+4x_2+x_3=8$ $4x_1+3x_2+x_3=12$ $x_2+x_3+6x_3=8$
3. $2x_1+6x_2+2x_3=15$

$4x_1+4x_2+3x_3=11$ $2x_2+x_3+7x_3=18$
4. $6x_2+2x_3=12$ $4x_1+5x_2+3x_3=9$ $2x_2+3x_3+7x_3=3$
5. $2x_1+x_2+3x_3=18$ $3x_1+x_2+6x_3=12$ $4x_2+6x_3+x_3=12$
6. $1x_1+4x_2+3x_3=22$ $3x_1+x_2+6x_3=12$ $2x_2+6x_3+x_3=14$
7. $2x_1+3x_2+4x_3=21$ $x_1+3x_2+x_3=12$ $3x_2+5x_3+7x_3=14$
8. $2x_1+3x_2=16$ $2x_1+3x_2+2x_3=14$ $x_2+5x_3+7x_3=11$
9. $2x_1+3x_3=12$ $2x_1+3x_2+2x_3=14$ $x_2+5x_3+7x_3=11$
10. $3x_1+3x_2+3x_3=17$ $2x_1+3x_2+2x_3=14$

$$x_2 + 5x_3 + 7x_3 = 11$$

Контрольные вопросы

1. В чем отличие массива NumPy от списка?
2. Можно ли изменить размер массива NumPy?
3. Как подключается библиотека NumPy?
4. Что выполняет метод dot библиотеки NumPy?

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Алексеев, В.В. – Теория алгоритмов. Учебно-методическое пособие. СарФТИ НИЯУ МИФИ, 2021 г. – 99 с.
2. Северенс, Ч. Введение в программирование на Python / Ч. Северенс. - 2-е изд., испр. - Москва : Национальный Открытый Университет «ИНТУИТ», 2016. - 231 с. : схем., ил. - <http://biblioclub.ru/>, экземпляров неограничено
3. Борзунов, С. В. Алгебра и геометрия с примерами на Python
Электронный ресурс / Борзунов С. В., Кургалин С. Д. : учебное пособие для вузов. - Санкт-Петербург : Лань, 2020. - 444 с. - ISBN 978-5-8114-5489-1, экземпляров неограничено
4. Лыкин, А. В. Математическое моделирование электрических систем и их элементов : Учебное пособие / Лыкин А. В. – Новосибирск : Новосибирский государственный технический университет, 2013. – 227 с. – Книга находится в базовой версии ЭБС IPRbooks. – ISBN 978-5-7782-2262-5
5. Герасименко, А. А. Передача и распределение электрической энергии : учеб. пособие / А. А. Герасименко, В. Т. Федин. – Ростов н/Д : Феникс ; Красноярск : Издательские проекты, 2006. – 720 с. : ил. – (Высшее образование). – Библиогр.: с. 667-671. – ISBN 5-222-08485-X (Феникс). – ISBN 5-98399-023-3 (Издательские проекты)
6. Электрические системы. : Математические задачи электроэнергетики : учебник для вузов / [В. А. Веников, Э. Н. Зуев, И. В. Литкенс и др.] ; под ред. В. А. Веникова. - Изд. 2-е, перераб. и доп. - М. : Высшая школа, 1981. - 288 с. : ил. - Гриф: Доп. МО для электроэнергетических спец. - Библиогр: с. 285-286
7. Луридас, П. Алгоритмы для начинающих: теория и практика для разработчика / Панос Луридас; [пер. с англ. Е.М. Егоровой]. – Москва: Эксмо, 2018. – 608 с.

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ**
**Федеральное государственное автономное образовательное учреждение
высшего образования**
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

**по организации самостоятельной работы студента
по дисциплине**
«Алгоритмизация и программирование»

Направление подготовки 13.03.02 «Электроэнергетика и
электротехника»
Направленность (профиль):
«Цифровые технологии в электроэнергетике»
Форма обучения: очная

Ставрополь, 2025

Введение

Самостоятельная работа является внеаудиторной и предназначена для самостоятельного изучения определенных тем дисциплины по материалам, рекомендованным преподавателем, а также в порядке подготовки к практическим занятиям.

Цель дисциплины «Алгоритмизация и программирование» – формирование согласно ООП ВО набора компетенций будущего бакалавра по направлению 13.03.02 «Электроэнергетика и электротехника».

Для достижения этой цели в процессе изучения дисциплины решаются следующие задачи: ознакомление с основными понятиями и терминами изучаемой дисциплины, которыми будущий специалист будет оперировать в своей практической деятельности.

1 Общая характеристика самостоятельной работы студента при изучении дисциплины

Основная задача высшего образования заключается в формировании творческой личности специалиста, способного к саморазвитию, самообразованию, инновационной деятельности. Решение этой задачи вряд ли возможно только путем передачи знаний в готовом виде от преподавателя к студенту. Для решения этой задачи в учебные планы всех специальностей включена самостоятельная работа, составляющая 50% учебного времени студента.

В широком смысле под самостоятельной работой студентов следует понимать совокупность всей самостоятельной деятельности студентов, как в учебной аудитории, так и вне ее, в контакте с преподавателем и в его отсутствие.

В учебном процессе выделяют два вида самостоятельной работы:

- аудиторная;
- внеаудиторная.

Аудиторная самостоятельная работа по дисциплине выполняется на учебных занятиях под непосредственным руководством преподавателя и по его заданию.

Внеаудиторная самостоятельная работа выполняется студентом по заданию преподавателя, но без его непосредственного участия.

Содержание внеаудиторной самостоятельной работы определяется в соответствии с рекомендуемыми видами заданий согласно примерной и рабочей программ учебной дисциплины. Видами заданий для внеаудиторной самостоятельной работы по дисциплине «Систему управления и сбора данных» являются:

- для овладения знаниями: чтение текста (учебника, первоисточника, дополнительной литературы), составление плана текста, графическое

изображение структуры текста, конспектирование текста, выписки из текста, работа со словарями и справочниками, ознакомление с нормативными документами, учебно-исследовательская работа, использование аудио- и видеозаписей, компьютерной техники и Интернета и др.;

– для закрепления и систематизации знаний: обработка текста, повторная работа над учебным материалом (учебника, первоисточника, дополнительной литературы, аудио и видеозаписей, составление плана, составление таблиц для систематизации учебного материала, ответ на контрольные вопросы, заполнение рабочей тетради, аналитическая обработка текста (аннотирование, рецензирование, реферирование, конспект-анализ и др);

– для закрепления навыков – подготовка к лабораторным работам, оформление отчетов по лабораторным работам, выполнение дополнительных заданий, выданных преподавателем.

Самостоятельная работа может осуществляться индивидуально или группами студентов в зависимости от цели, объема, конкретной тематики самостоятельной работы, уровня сложности, уровня умений студентов. Контроль результатов внеаудиторной самостоятельной работы студентов может осуществляться в пределах времени, отведенного на обязательные учебные занятия по дисциплине и внеаудиторную самостоятельную работу студентов по дисциплине, может проходить в письменной, устной или смешанной форме.

Следует с первых дней обучения в вузе приучать себя к целесообразному распределению занятий по месяцам и неделям семестра. При этом очень важна равномерная работа как основное условие успешных занятий в вузе.

Высшей ступенью обучения является приобретение навыков самостоятельной творческой работы, а творческому применению знаний нужно учиться параллельно с их приобретением.

Задачами самостоятельной работы студентов по дисциплине «Систему управления и сбора данных» являются:

- формирования готовности студентов к поиску, обработке и применению информации для решения профессиональных задач;
- систематизация и закрепление полученных теоретических знаний и практических умений студентов;
- развитие познавательных способностей и активности студентов, творческой инициативы, самостоятельности, ответственности и организованности;
- формирование самостоятельности мышления, способностей к саморазвитию, самосовершенствованию и самореализации;
- выработка навыков эффективной самостоятельной профессиональной деятельности.

2 План-график выполнения самостоятельной работы

Общий объём самостоятельной работы в рамках изучения дисциплины «Алгоритмизация и программирование» составляет 54 астрономических часа.

В структуре самостоятельной работы предусмотрено самостоятельное изучение теоретического материала, подготовка к лабораторным занятиям.

В таблице 2.1 приведен план график выполнения самостоятельной работы.

Таблица 2.1 – План график выполнения самостоятельной работы

Коды реализуемых компетенций, индикатора(ов)	Вид деятельности студентов	Средства и технологии оценки	Объем часов, в том числе		
			СРС	Контактная работа с преподавателем	Всего
3 семестр					
ИД-5 _{ПК-4}	Подготовка к лекции	Собеседование	8.00	1.00	9.00
ИД-5 _{ПК-4}	Подготовка к практическим занятиям	Собеседование	19.00	1.00	20.00
ИД-5 _{ПК-4}	Самостоятельное изучение литературы	Тестирование	23.00	2.00	25.00
Итого за 3 семестр			50.00	4.00	54.00
Итого			50.00	4.00	54.00

3 Методические рекомендации по изучению теоретического материала

Большая часть самостоятельной работы студента состоит в изучении литературы. Одна из задач студента – научиться самостоятельно работать с книгой, а это требует определенных затрат энергии и времени. Самостоятельная работа с книгой может быть успешной, если текст не только прочитан, но и законспектирован. Существует несколько форм записей, но любая форма записи не даст нужного результата, если не будет пробуждать мысли того, кто ее ведет, если отсутствует активная работа ума и формирование своих выводов из прочитанного.

Выбор формы записи зависит от индивидуальных особенностей человека, его образованности и опыта. При этом не меньшую роль играет назначение записей, то есть то, какие задачи ставит перед собой человек (для самообразования, для выступления на семинаре, для использования в будущем).

Введение записей мобилизует наряду со зрительной памятью, также и моторную память. Кроме того, у человека, систематически ведущего записи изучаемой литературы, создается свой фонд материалов для быстрого повторения и мобилизации накопленных знаний.

Все записи должны быть убористыми и компактными. Интервалы между строками должны быть достаточными, чтобы вписывать дополнения. Рекомендуются вести записи ручкой, а карандашом или ручкой другого цвета пользоваться для отметок и выделений при последующей работе. Полезно также датировать записи.

Записи могут носить различный характер: план, выписки, тезисы, аннотирование, конспектирование.

1. План - наиболее краткая формой записи. Это перечень вопросов, рассматриваемых в книге или статье. План обычно раскрывает структуру

произведения, логику автора, способствует лучшей ориентации в содержании.

Так, составленным планом можно воспользоваться, чтобы вспомнить прочитанное или быстро отыскать в книге нужное место. Представление об основных пунктах плана дает оглавление книги, поэтому во многих случаях наименования глав и разделов можно использовать в качестве пунктов. Составление плана приучает логически мыслить, вырабатывать умение сжато и последовательно излагать суть вопроса в письменной и устной форме.

Существует два способа составления плана: работа над ним по ходу чтения и составление плана после ознакомления с произведением. При этом план получается более последовательным и стройным.

Трудность составления плана состоит в том, что надо выяснить для себя, прежде всего, построение изучаемого текста, ход мыслей автора и лишь затем изложить содержание работы кратко и ясно. При всей своей краткости план дает представление о содержании прочитанного. Следует учесть, что форма плана не исключает цитирования отдельных мест и обобщений. Различают простой и развернутый план. В отличие от простого плана развернутый план не только содержит перечисление вопросов, но и раскрывает основные идеи произведения, может включать выдержки из него, схемы, таблицы. Планом, особенно развернутым, необходимо пользоваться при написании выступления или статьи.

В целом развернутый план дает гораздо большее представление о произведении, его основных идеях, задачах, которые в нем решаются. Он может включать положения, замечания, собственные мысли студента.

Важно знать, что составление планов помогает вырабатывать способность к отвлеченному, абстрактному мышлению, но наибольшую пользу составление плана даст подготовленным лицам, которые бывают достаточно лишь взглянуть на перечень основных вопросов, чтобы воспроизвести содержание прочитанного.

2. Тезисы – более сложная и совершенная форма записи, чем составление плана.

Это сжатое изложение основных мыслей прочитанного произведения или подготовляемого вступления. Особенностью тезисов является их утвердительный характер.

В них сосредотачивается самое главное, только выводы и обобщения, в них меньше доказательств, иллюстрации и пояснений. Тезисы не должны повторять дословно текст, но в ряде мест могут быть близки к нему, воспроизводя некоторые характерные выражения автора, важные для понимания хода его мыслей. Составление тезисов помогает глубже понять основные идеи произведения, выделить главное в нем; приучают сжато, точно и четко сформулировать свои мысли, повышает культуру речи и письма. При составлении тезисов учитывают следующее. Прежде всего, если произведение небольшое, необходимо внимательно изучать его в целом, если большое – изучать по главам и разделам. Затем, когда будут ясны основные идеи, кратко и последовательно излагать их в виде пунктов.

Различают простые и сложные, развернутые тезисы. Если записывают только утверждение чего – либо, такой тезис называют простым, а сложным тезисом будет выражение главной мысли, содержащее, кроме утверждения, еще и краткое ее доказательство.

Часто тезисы формулируются самим автором как выводы и обобщения в заключении книги или разделах книги. Нередко тезисы выделяются в тексте другим шрифтом.

Рекомендуется делать тезисные записи своими словами, причем можно записывать один абзац за другим, учитывая смысловую связь между ними. Но в большинстве случаев следует составлять сводный тезис, сложный по форме. При этом объединяется несколько утверждений, тесно связанных между собой.

Тезисы по содержанию очень близки к конспекту, но конспект носит более описательный характер, и его положения не столь категоричны, как в тезисах. Кроме того, конспект представляет собой более полную форму записи.

Следует отметить, что различие между формами записей условно, но в любой форме запись – важнейшая часть самостоятельной работы с книгой.

3. Выписки. Это записи текста из книги: теоретических положений, статистических данных, имеющих по мнению читателя важное значение.

Достоинство выписок состоит в точности воспроизведения текста книги, удобстве пользования записями при последующей работе, в накоплении обобщений и фактического материала. Выписки полезны для повторения, освежения в памяти прочитанного, для быстрой мобилизации своих знаний, когда необходимо в короткий срок вспомнить материал. Выписки выделяют из текста самое главное и тем самым помогают глубже понять его. Без них трудно обойтись при подготовке доклада, реферата, выступления. Выписки следует рассматривать как составную часть тезисов и конспектов.

Выписывать текст можно и по ходу чтения и после его завершения. В последнем случае надо замечать места, которые потом будут выписаны. Необходимо каждую выписку снабжать ссылкой на источник с указанием соответствующей страницы. Это нужно, чтобы в последствии можно было быстро найти в книге соответствующее место. Целесообразно выписывать из текста только такие места, в которых содержится самое главное, суть вопроса. Выписки должны быть ориентированы на изучение произведения в целом, а не отдельных мест, поскольку положения, вырванные из общего контакта, понимаются нередко совсем не так, как этого хотел автор. Иначе говоря, отдельно взятые, лишенные пояснений выдержки могут быть не поняты или поняты неправильно.

Выписки бывают дословные (цитаты) и «свободные», когда мысли автора излагаются своими словами. Следует учесть, что большие отрывки, которые трудно цитировать, целесообразнее в краткой форме переложить своими словами, но «яркие» и важные места лучше выписывать дословно. Каждую цитату следует заключать в кавычки. Если ее берут из середины предложения, то после вводных кавычек ставят три точки. Ставят их и в конце цитаты, если из предложения опущены последние слова.

Следует знать, что какого-либо единого метода выписок, годного для всех случаев, не существует, поскольку у каждого человека свои особенности мышления и восприятия, свой подход к теме. Все это влияет на содержание и характер выписок.

Выписки рекомендуется хранить в картотеке, конвертах или папках, на которых следует обозначить общую тему.

4. Аннотация – еще одна форма записи, являющаяся кратким обобщением содержания книги. Ею удобно пользоваться, если имеется намерение вернуться к изучаемому произведению. Аннотация может быть необходима и для того, чтобы не забыть о нем.

Для составления аннотации надо сначала полностью прочитать и глубоко продумать произведение. При всей своей краткости аннотация может содержать отдельные фрагменты авторского текста, а не только оценку книги или статьи.

5. Резюме очень близко к аннотации. Это запись, являющаяся краткой оценкой прочитанного материала. Различие между ними состоит в том, что аннотация сжато характеризует произведение в целом, а резюме концентрирует внимание на его выводах, главных итогах.

6. Конспект – наиболее совершенная и наиболее сложная форма записи. Слово «конспект» происходит от латинского «conspectus», что означает «обзор, изложение». В правильно составленном конспекте обычно выделено самое основное в изучаемом тексте, сосредоточено внимание на

наиболее существенном, в кратких и четких формулировках обобщены важные теоретические положения.

Конспект представляет собой относительно подробное, последовательное изложение содержания прочитанного. На первых порах целесообразно в записях ближе держаться тексту, прибегая зачастую к прямому цитированию автора. В дальнейшем, по мере выработки навыков конспектирования, записи будут носить более свободный и сжатый характер.

Конспект книги обычно ведется в тетради. В самом начале конспекта указывается фамилия автора, полное название произведения, издательство, год и место издания. При цитировании обязательная ссылка на страницу книги. Если цитата взята из собрания сочинений, то необходимо указать соответствующий том. Следует помнить, что четкая ссылка на источник – непереносимое правило конспектирования. Если конспектируется статья, то указывается, где и когда она была напечатана.

Конспект подразделяется на части в соответствии с заранее продуманным планом. Пункты плана записываются в тексте или на полях конспекта. Писать его рекомендуется четко и разборчиво, так как небрежная запись с течением времени становится малопонятной для ее автора. Существует правило: конспект, составленный для себя, должен быть по возможности написан так, чтобы его легко прочитал и кто-либо другой.

Формы конспекта могут быть разными и зависят от его целевого назначения (изучение материала в целом или под определенным углом зрения, подготовка к докладу, выступлению на занятии и т.д.), а также от характера произведения (монография, статья, документ и т.п.). Если речь идет просто об изложении содержания работы, текст конспекта может быть сплошным, с выделением особо важных положений подчеркиванием или различными значками.

В случае, когда не ограничиваются переложением содержания, а фиксируют в конспекте и свои собственные суждения по данному вопросу

или дополняют конспект соответствующими материалами их других источников, следует отводить место для такого рода записей. Рекомендуется разделить страницы тетради пополам по вертикали и в левой части вести конспект произведения, а в правой свои дополнительные записи, совмещая их по содержанию.

Конспектирование в большей мере, чем другие виды записей, помогает вырабатывать навыки правильного изложения в письменной форме важные теоретических и практических вопросов, умение четко их формулировать и ясно излагать своими словами.

Таким образом, составление конспекта требует вдумчивой работы, затраты времени и труда. Зато во время конспектирования приобретаются знания, создается фонд записей.

Конспект может быть текстуальным или тематическим. В текстуальном конспекте сохраняется логика и структура изучаемого произведения, а запись ведется в соответствии с расположением материала в книге. За основу тематического конспекта берется не план произведения, а содержание какой-либо темы или проблемы.

Текстуальный конспект желательно начинать после того, как вся книга прочитана и продумана, но это, к сожалению, не всегда возможно. В первую очередь необходимо составить план произведения письменно или мысленно, поскольку в соответствии с этим планом строится дальнейшая работа. Конспект включает в себя тезисы, которые составляют его основу. Но, в отличие от тезисов, конспект содержит краткую запись не только выводов, но и доказательств, вплоть до фактического материала. Иначе говоря, конспект – это расширенные тезисы, дополненные рассуждениями и доказательствами, мыслями и соображениями составителя записи.

Как правило, конспект включает в себя и выписки, но в него могут войти отдельные места, цитируемые дословно, а также факты, примеры, цифры, таблицы и схемы, взятые из книги. Следует помнить, что работа над

конспектом только тогда будет творческой, когда она не ограничена текстом изучаемого произведения. Нужно дополнять конспект данными из другими источников.

В конспекте необходимо выделять отдельные места текста в зависимости от их значимости. Можно пользоваться различными способами: подчеркиваниями, вопросительными и восклицательными знаками, репликами, краткими оценками, писать на полях своих конспектов слова: «важно», «очень важно», «верно», «характерно».

В конспект могут помещаться диаграммы, схемы, таблицы, которые придадут ему наглядность.

Составлению тематического конспекта предшествует тщательное изучение всей литературы, подобранной для раскрытия данной темы. Бывает, что какая-либо тема рассматривается в нескольких главах или в разных местах книги. А в конспекте весь материал, относящийся к теме, будет сосредоточен в одном месте. В плане конспекта рекомендуется делать пометки, к каким источникам (вплоть до страницы) придется обратиться для раскрытия вопросов. Тематический конспект составляется обычно для того, чтобы глубже изучить определенный вопрос, подготовиться к докладу, лекции или выступлению на семинарском занятии. Такой конспект по содержанию приближается к реферату, докладу по избранной теме, особенно если включает и собственный вклад в изучение проблемы.

45 Методические указания по подготовке к практическим занятиям

В рамках самостоятельной работы предусмотрена подготовка студента к практическим занятиям. Практические занятия позволяют интегрировать теоретические знания и формировать практические умения и навыки студентов в процессе учебной деятельности.

Цели практических занятий по дисциплине «Алгоритмизация и программирование»:

- 1) закрепление теоретического материала путем систематического контроля за самостоятельной работой студентов;
- 2) формирование умений использования теоретических знаний в процессе выполнения практических работ;
- 3) развитие аналитического мышления путем обобщения результатов практических работ;
- 4) формирование навыков оформления результатов практических работ в виде таблиц, графиков, выводов.

На практических занятиях осуществляются следующие формы работ со студентами: индивидуальная (оценка знаний, выполненных тестовых заданий, проверка рабочих тетрадей); групповая (выполнение заданий малыми группами по 2-4 человека); фронтальная (подведение итогов выполнения лабораторных работ).

Список рекомендуемой литературы

Основная литература:

1. Алексеев, В.В. – Теория алгоритмов. Учебно-методическое пособие. СарФТИ НИЯУ МИФИ, 2021 г. – 99 с.

2. Северенс, Ч. Введение в программирование на Python / Ч. Северенс. - 2-е изд., испр. - Москва : Национальный Открытый Университет «ИНТУИТ», 2016. - 231 с. : схем., ил. - <http://biblioclub.ru/>, экземпляров неограничено

3. Борзунов, С. В. Алгебра и геометрия с примерами на Python Электронный ресурс / Борзунов С. В., Кургалин С. Д. : учебное пособие для вузов. - Санкт-Петербург : Лань, 2020. - 444 с. - ISBN 978-5-8114-5489-1, экземпляров неограничено

4. Лыкин, А. В. Математическое моделирование электрических систем и их элементов : Учебное пособие / Лыкин А. В. – Новосибирск : Новосибирский государственный технический университет, 2013. – 227 с. – Книга находится в базовой версии ЭБС IPRbooks. – ISBN 978-5-7782-2262-5

Дополнительная литература:

1. Герасименко, А. А. Передача и распределение электрической энергии : учеб. пособие / А. А. Герасименко, В. Т. Федин. – Ростов н/Д : Феникс ; Красноярск : Издательские проекты, 2006. – 720 с. : ил. – (Высшее образование). – Библиогр.: с. 667-671. – ISBN 5-222-08485-X (Феникс). – ISBN 5-98399-023-3 (Издательские проекты)

2. Электрические системы. : Математические задачи электроэнергетики : учебник для вузов / [В. А. Веников, Э. Н. Зуев, И. В. Литкенс и др.] ; под ред. В. А. Веникова. - Изд. 2-е, перераб. и доп. - М. : Высшая школа, 1981. - 288 с. : ил. - Гриф: Доп. МО для электроэнергетических спец. - Библиогр: с. 285-286

3. Луридас, П. Алгоритмы для начинающих: теория и практика для разработчика / Панос Луридас; [пер. с англ. Е.М. Егоровой]. – Москва: Эксмо, 2018. – 608 с.

Учебно-методическая литература:

6 Алгоритмизация и программирование. Методические указания по выполнению практических занятий. Направление подготовки 13.03.02 «Электроэнергетика и электротехника», СКФУ, 2024 /сост. Кононова Н.Н.

Интернет-ресурсы:

7. <http://rdsn.org>

8. <https://msdn.microsoft.com/ru-ru/>

9. <https://stepik.org>