

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

Николаев Е.И.

СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Курс лекций

Ставрополь, 2024

Содержание

Лекция 1. Технологии программирования.....	7
1.1. Аппаратное и программное обеспечение компьютера	7
1.1.1. Центральный процессор.....	8
1.1.2. Оперативная память.....	9
1.1.3. Вторичные устройства хранения.....	10
1.1.4. Устройства ввода	11
1.1.5. Устройства вывода.....	11
1.1.6. Программное обеспечение.....	12
1.1.7. Системное программное обеспечение	12
1.1.8. Прикладное программное обеспечение	13
1.2. Представление данных в памяти компьютера. Кодирование	13
1.3. Выполнение программ	19
1.4. Компиляторы и интерпретаторы. Языки программирования	23
Вопросы для самопроверки по теме 1	29
Лекция 2. Принципы разработки программ	31
2.1. Средства разработки Python	31
2.2. Структура скрипта Python.....	32
2.3. Цикл проектирования программы.....	34
2.4. Псевдокод. Блок-схема.....	36
2.5. Обработка и вывод данных.....	39
2.5.1. Вывод данных на экран при помощи функции print.....	40
2.5.2. Комментарии	42
2.6. Переменные. Ввод данных.....	43
2.6.1. Создание переменных инструкцией присваивания.....	44
2.6.2. Правила именования переменных.....	46
2.6.3. Вывод нескольких значений при помощи функции print.....	48
2.6.4. Повторное присваивание значений переменным	49
2.6.5. Числовые типы данных и числовые литералы	50
2.6.6. Повторное присвоение переменной значения другого типа	51

2.6.7. Ввод данных	52
2.6.8. Чтение чисел при помощи функции <code>input</code>	53
2.7. Выполнение расчетов	55
Вопросы для самопроверки по теме 2	57
Лекция 3. Управление ходом выполнения программы.....	60
3.1. Условный оператор.....	60
3.1.1. Булевые выражения и операторы сравнения	63
3.1.2. Инструкция <code>if-else</code>	64
3.1.3. Вложенные структуры принятия решения и инструкция <code>if-elif-else</code>	66
3.2. Структуры повторения	69
3.2.1. Цикл с условием повторения	70
3.2.2. Цикл <code>for</code> : цикл со счетчиком повторений	73
3.3. Вычисления с использованием циклов.....	76
Вопросы для самопроверки по теме 3	78
Лекция 4. Функции	82
4.1. Процедуры и функции	82
4.2. Объявление и использование функций	85
4.3. Область действия и локальные переменные	87
4.4. Передача аргументов в функцию	88
4.5. Возврат значений из функции	90
4.6. Функции стандартной библиотеки и инструкция <code>import</code>	92
4.7. Рекурсия	94
4.8. Функции в вычислительных задачах	95
Вопросы для самопроверки по теме 4	97
Лекция 5. Файлы и исключения	100
5.1. Основы файлового ввода-вывода	100
5.1.1. Типы файлов	103
5.1.2. Методы доступа к файлам	103
5.1.3. Имена файлов и файловые объекты	104

5.1.4. Открытие файла	105
5.1.5. Указание места расположения файла	106
5.2. Запись данных в файл.....	107
5.3. Чтение данных из файла.....	108
5.4. Применение циклов для обработки файлов	109
5.5. Обработка исключительных ситуаций	111
Вопросы для самопроверки по теме 5	116
Лекция 6. Списки	118
6.1. Изменяемые и неизменяемые последовательности.	118
6.2. Список. Индексация.	119
6.2.1. Оператор повторения.....	120
6.2.2. Последовательный обход списка в цикле for	121
6.2.3. Индексация	123
6.2.4. Функция len	124
6.3. Изменение элементов списка.....	125
6.4. Поиск элементов списка.....	127
6.4.1. Срезы	127
6.4.2. Поиск значений в списках при помощи инструкции in	128
6.5. Операции над списками. Методы списков.....	129
Вопросы для самопроверки по теме 6	130
Лекция 7. Строки.....	134
7.1. Основы работы со строками	134
7.1.1. Обход строкового значения в цикле for	134
7.1.2. Индексация	135
7.1.3. Проверка строковых значений при помощи in и not in	136
7.2. Строковые методы	136
7.2.1. Методы проверки строковых значений	137
7.2.2. Методы модификации строковых значений	138
7.2.3. Поиск и замена	139
7.2.4. Разбиение строкового значения	140

7.3. Чтение CSV-файлов.....	141
Вопросы для самопроверки по теме 7	142
Лекция 8. Коллекции	146
8.1. Кортежи.....	146
8.2. Словари	147
8.2.1. Создание словаря	147
8.2.2. Получение значений из словаря	148
8.2.3. Применение операторов in и not in для проверки на наличие значения в словаре.....	149
8.2.4. Добавление, удаление и изменение значений в словаре	150
8.2.5. Словарные методы.....	150
8.3. Множества	151
8.3.1. Добавление и удаление элементов.....	153
8.3.2. Применение цикла for для обхода множества	153
8.3.3. Операции над множествами	154
8.4. Сериализация объектов	155
Вопросы для самопроверки по теме 8	157
Лекция 9. Многомодульные программы	162
9.1. Модули и клиенты	162
9.2. Стандартная библиотека Python. Установка библиотек	165
9.3. Модуль os.....	168
9.4. Модуль math	170
9.5. Модуль itertools	173
9.6. Модуль random	176
9.7. Пользовательские модули	179
Вопросы для самопроверки по теме 9	181
Лекция 10. Объектно-ориентированное программирование.....	182
10.1. Процедурное и объектно-ориентированное.....	182
10.1.1. Возможность многократного использования объекта	185
10.1.2. Пример объекта из повседневной жизни	186

10.2. Класс.....	187
10.2.1. Определения классов.....	190
10.2.2. Скрытие атрибутов	194
10.3. Объект. Экземпляры класса.....	196
10.3.1. Работа с экземплярами	197
10.3.2. Методы-получатели и методы-мутаторы.....	199
10.3.3. Метод <code>__str__</code>	199
10.4. Консервация пользовательских объектов	201
Вопросы для самопроверки по теме 10	202
Лекция 11. Анонимные функции	204
11.1. Лямбда-выражения	204
11.2. Применение анонимных функций в алгоритмах сортировки, фильтрации и отображения.....	205
11.2.1. Применение лямбда-выражений для сортировки.....	205
11.2.2. Применение лямбда-выражений для фильтрации.....	206
11.2.3. Применение лямбда-выражений для отображения	207
Вопросы для самопроверки по теме 11	209
Лекция 12. Визуализация данных	210
12.1. Манипулирование данными средствами pandas.....	210
12.2. Построение графиков и диаграмм.....	214
12.2.1. Визуализация количественных признаков	214
12.2.2. Категориальные признаки.....	218
12.2.3. Визуализация соотношения количественных признаков .	219
Вопросы для самопроверки по теме 12	223
СПИСОК ЛИТЕРАТУРЫ	224

Лекция 1. Технологии программирования.

План лекции

1. Аппаратное и программное обеспечение компьютера.
2. Представление данных в памяти компьютера. Кодирование.
3. Выполнение программ.
4. Компиляторы и интерпретаторы. Языки программирования.

1.1. Аппаратное и программное обеспечение компьютера

Физические устройства, из которых состоит компьютер, называются аппаратным обеспечением компьютера. Работающие на компьютере программы называются программным обеспечением.

Термин «аппаратное обеспечение» (hardware) относится ко всем физическим устройствам или компонентам, из которых состоит компьютер. Компьютер представляет собой не единое устройство, а систему устройств, которые работают во взаимодействии друг с другом.

Как показано на рис. 1.1, типичная компьютерная система состоит из следующих главных компонентов:

- ◆ центрального процессора (ЦП);
- ◆ основной памяти;
- ◆ вторичных устройств хранения данных;
- ◆ устройств ввода;
- ◆ устройств вывода.

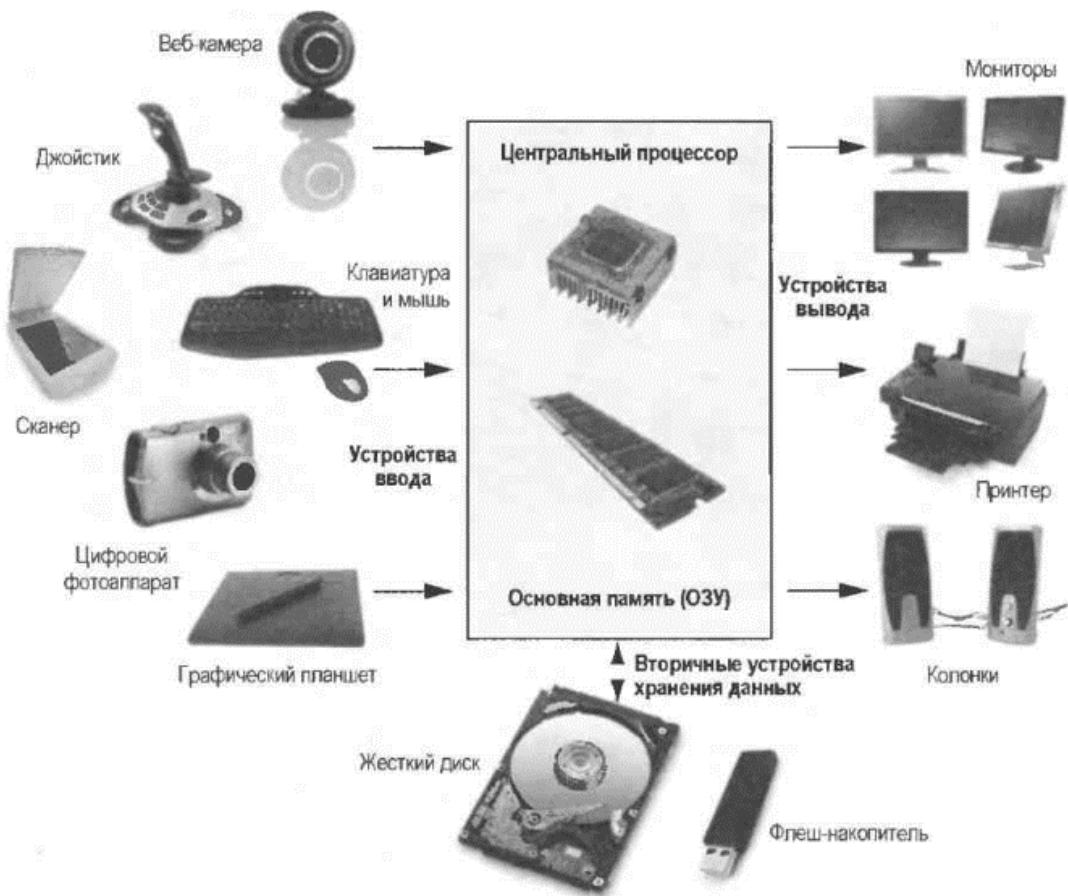


Рисунок 1.1 – Типовые компоненты компьютерной системы

1.1.1. Центральный процессор

Когда компьютер занят задачами, которые программа поручает ему выполнить, мы говорим, что компьютер выполняет, или исполняет, программу. Центральный процессор (ЦП) является той частью компьютера, которая фактически исполняет программы. Это самый важный компонент в компьютере, потому что без него компьютер не сможет выполнять программы.

В самых ранних компьютерах ЦП представлял собой огромные устройства, состоящие из электрических и механических компонентов, таких как вакуумные лампы и переключатели.

В наше время ЦП представляют собой миниатюрные кристаллы интегральной микросхемы, которые называются микропроцессорами. На рис. 1.2 представлена фотография современного центрального процессора.



Рисунок 1.2 – Центральный процессор компьютера

1.1.2. Оперативная память

Основную память можно представить, как рабочую область компьютера. Это то место, где компьютер хранит программу, пока та выполняется, и данные, с которыми программа работает. Например, предположим, вы используете программу обработки текста, чтобы написать работу для одного из своих учебных занятий. Пока вы это делаете, программа обработки текста и ваше сочинение хранятся в основной памяти.

Основную память принято называть оперативным запоминающим устройством (ОЗУ), или запоминающим устройством с произвольным доступом (ЗУПД), или просто оперативной памятью. Называется она так, потому что ЦП способен получать немедленный доступ к данным, хранящимся в любой произвольной единице хранения в ОЗУ. Как правило, ОЗУ является энергозависимым типом памяти, которая используется только для временного хранения, пока программа работает. Когда компьютер выключают, содержимое ОЗУ стирается.

В вашем компьютере ОЗУ размещено в кристаллах интегральной микросхемы, подобных тем, которые показаны на рис. 1.3.

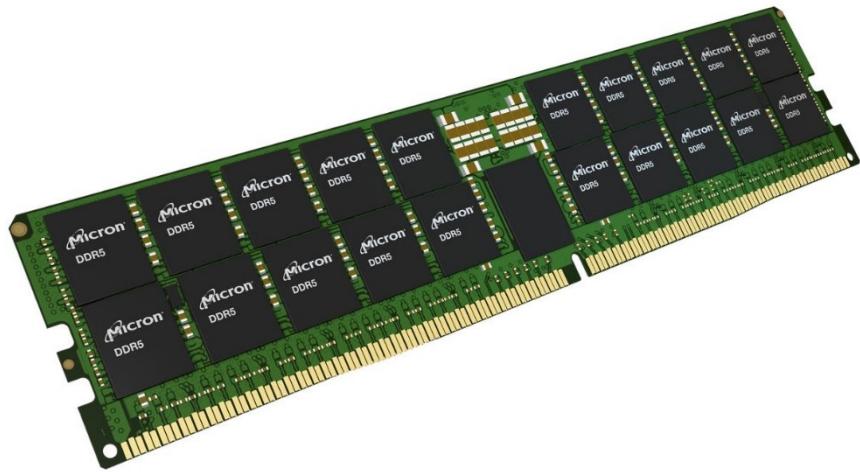


Рисунок 1.3 – Модуль основной памяти компьютера

1.1.3. Вторичные устройства хранения

Вторичное устройство хранения – это тип памяти, который может хранить данные в течение долгих промежутков времени, даже когда к компьютеру не подведено электропитание.

В обычных условиях программы хранятся во вторичной памяти и загружаются в основную память по мере необходимости. Важные данные, такие как документы текстового редактора, данные платежных ведомостей и складских запасов, тоже хранятся во вторичной памяти.

Наиболее распространенным типом вторичного устройства хранения является жесткий диск. Традиционный жесткий диск сохраняет данные путем их магнитной записи на вращающийся круговой диск. Все большую популярность приобретают твердотельные диски, которые сохраняют данные в твердотельной памяти. Твердотельный диск не имеет подвижных частей и работает быстрее, чем традиционный диск. В большинстве компьютеров обязательно имеется какое-то вторичное устройство хранения, традиционный диск или твердотельный диск, смонтированное внутри своего корпуса. Имеются также вторичные устройства хранения, которые присоединяются к одному из коммуникационных портов компьютера.

Вторичные устройства хранения используются для создания резервных копий важных данных либо для перемещения данных на другой компьютер.

Для копирования данных и их перемещения на другие компьютеры помимо вторичных устройств хранения были созданы разнообразные типы устройств. Возьмем, к примеру, USB-диски. Эти небольшие устройства подсоединяются к порту USB (универсальной последовательной шине) компьютера и определяются в системе как внешний жесткий диск. Правда, эти диски на самом деле не содержат дисковых пластин. Они хранят данные в памяти особого типа – во флеш-памяти. USB-диски, именуемые также картами памяти и флеш-накопителями, имеют небольшую стоимость, надежны и достаточно маленькие, чтобы можно было носить их в кармане одежды.

1.1.4. Устройства ввода

Вводом называются любые данные, которые компьютер собирает от людей и от различных устройств. Компонент, который собирает данные и отправляет их в компьютер, называется устройством ввода. Типичными устройствами ввода являются клавиатура, мышь, сенсорный экран, сканер, микрофон и цифровая фотокамера. Дисководы и оптические диски можно тоже рассматривать как устройства ввода, потому что из них извлекаются программы и данные, которые затем загружаются в оперативную память компьютера.

1.1.5. Устройства вывода

Выводом являются любые данные, которые компьютер производит для людей или для различных устройств. Это может быть торговый отчет, список имен или графическое изображение. Данные отправляются в устройство вывода, которое их форматирует и предъявляет.

Типичными устройствами вывода являются видеодисплеи и принтеры. Дисководы можно тоже рассматривать как устройства вывода, потому что компьютерная система отправляет на них данные с целью их сохранения.

1.1.6. Программное обеспечение

Для функционирования компьютера требуется программное обеспечение. Все, что компьютер делает с момента нажатия на кнопку включения питания и до момента выключения компьютерной системы, находится под контролем программного обеспечения. Имеются две общие категории программного обеспечения: системное программное обеспечение и прикладное программное обеспечение. Большинство компьютерных программ четко вписывается в одну из этих двух категорий. Давайте рассмотрим на каждую из них подробнее.

1.1.7. Системное программное обеспечение

Программы, которые контролируют и управляют основными операциями компьютера, обычно называются системным программным обеспечением. К системному ПО, как правило, относятся следующие типы программ.

Операционная система (ОС) – фундаментальный набор программ на компьютере. Она управляет внутренними операциями аппаратного обеспечения компьютера и всеми подключенными к компьютеру устройствами, позволяет сохранять данные и получать их из устройств хранения, обеспечивает выполнение других программ. К популярным операционным системам для ноутбуков и настольных компьютеров относят Windows, Mac OS X и Linux. Популярные операционные системы для мобильных устройств – Android и iOS.

Обслуживающая программа, или утилита, выполняет специализированную задачу, которая расширяет работу компьютера или

гарантирует сохранность данных. Примерами обслуживающих, или сервисных, программ являются вирусные сканеры, программы сжатия файлов и программы резервного копирования данных.

Инструменты разработки программного обеспечения – это программы, которые программисты используют для создания, изменения и тестирования программного обеспечения.

Ассемблеры, компиляторы и интерпретаторы являются примерами программ, которые попадают в эту категорию.

1.1.8. Прикладное программное обеспечение

Программы, которые делают компьютер полезным для повседневных задач, называются прикладным программным обеспечением. На выполнение этих программ на своих компьютерах люди по обыкновению тратят большую часть своего времени.

1.2. Представление данных в памяти компьютера. Кодирование

Все хранящиеся в компьютере данные преобразуются в последовательности, состоящие из нулей и единиц.

Память компьютера разделена на единицы хранения, которые называются байтами. Одного байта памяти достаточно только для того, чтобы разместить одну букву алфавита или небольшое число. Для того чтобы делать что-то более содержательное, компьютер должен иметь очень много байтов. Большинство компьютеров сегодня имеет миллионы или даже миллиарды байтов оперативной памяти.

Каждый байт разделен на восемь меньших единиц хранения, которые называются битами, или разрядами. Термин «бит» происходит от англ. binary digit (двоичная цифра). В большинстве компьютерных систем биты – это крошечные электрические компоненты, которые могут содержать либо положительный, либо отрицательный заряд. Программисты рассматривают

положительный заряд, как переключатель в положении «Включено», и отрицательный заряд, как переключатель в положении «Выключено». На рис. 1.4 показано, как программист может рассматривать байт памяти: как коллекцию переключателей, каждый из которых установлен в положение «Вкл» либо «Выкл».

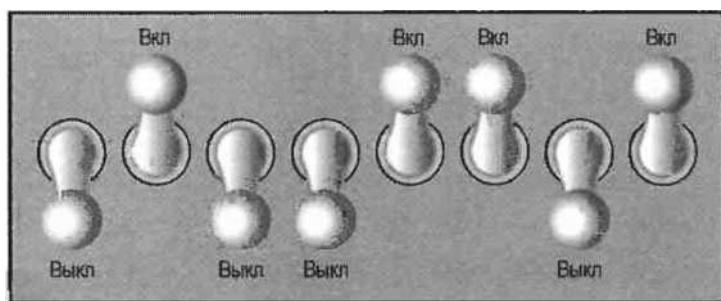


Рисунок 1.4 – Байт как набор переключателей

Когда порция данных сохраняется в байте, компьютер размещает восемь битов в двухпозиционной комбинации "включено-выключено", которая представляет данные. Например, на рис. 1.5, слева показано, как в байте будет размещено число 77, а справа – латинская буква А.

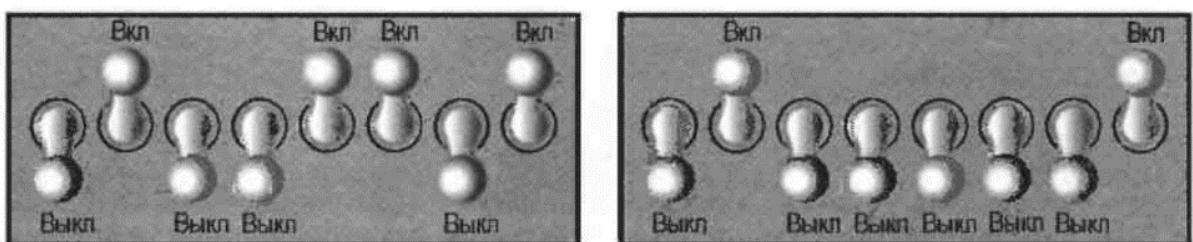


Рисунок 1.5 – Комбинации двоичных разрядов для числа 77 (слева) и латинской буквы А (справа)

Бит используется для представления чисел в весьма ограниченной форме. В зависимости от того, "включен" он или "выключен", бит может представлять одно из двух разных значений.

В компьютерных системах выключенный бит представляет число 0, а включенный – число 1. Это идеально соответствует двоичной системе исчисления, в которой все числовые значения записываются как последовательности нулей и единиц. Вот пример числа, которое записано в двоичной системе исчисления:

10011101

Позиция каждой цифры в двоичном числе соответствует определенному значению. Как показано на рис. 1.6, начиная с самой правой цифры и двигаясь влево, значения позиций равняются $2^0, 2^1, 2^2, 2^3$ и т. д.



Рисунок 1.6 – Значения разрядов двоичной записи числа в десятичной системе

Для того чтобы определить значение двоичного числа, нужно просто сложить позиционные значения всех единиц. Например, в двоичном числе 10011101 позиционные значения единиц равняются 1, 4, 8, 16 и 128 (рис. 1.7).

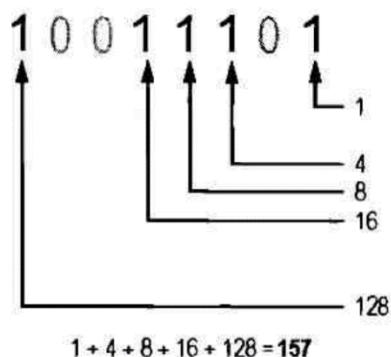


Рисунок 1.7 – Определение значения 10011101

Сумма всех этих позиционных значений равняется 157. Значит, двоичное число 10011101_2 равняется 157_{10} в десятичной системе исчисления.

На рис. 1.8 показано, как можно изобразить хранение числа 157 в байте оперативной памяти. Каждая 1 представлена битом в положении «Вкл», а каждый 0 – битом в положении «Выкл».

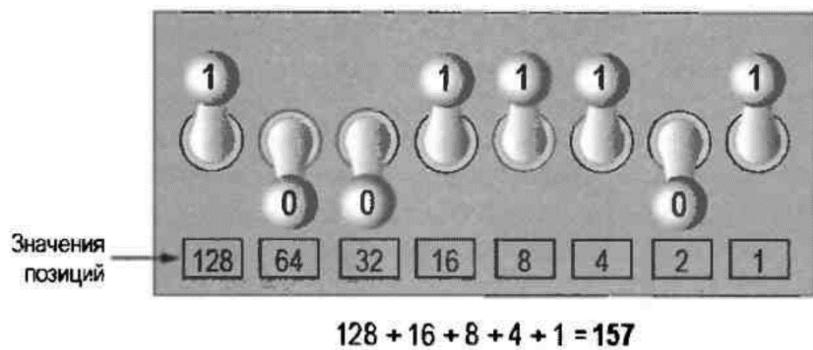


Рисунок 1.8 – Комбинация переключателей (двоичных разрядов) для числа 157.

Когда всем битам в байте назначены нули (т.е. они выключены), значение байта равняется 0. Когда всем битам в байте назначены единицы (т.е. они включены), байт содержит самое большое значение, которое в нем может быть размещено. Оно равняется $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$. Этот предел является следствием того, что в байте всего восемь бит.

А если нужно сохранить число, которое больше 255? Необходимо использовать еще один байт. В итоге получаем 16 бит. Позиционные значения этих 16 бит будут $2^0, 2^1, 2^2, 2^3$ и т. д. до 2^{15} . Максимальное значение, которое можно разместить в двух байтах, равно 65 535 (рис. 1.9). Если же нужно сохранить еще большее число, для этого потребуется больше байтов.

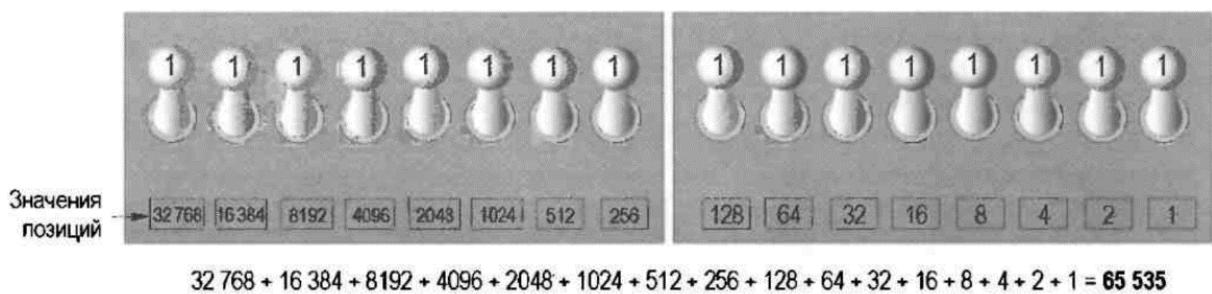


Рисунок 1.9 – Для большого числа использованы два байта.

Любая порция данных в оперативной памяти компьютера должна храниться как двоичное число. Это относится и к символам, таким как буквы и знаки препинания. Когда символ сохраняется в памяти, он сначала преобразуется в цифровой код. И затем этот цифровой код сохраняется в памяти как двоичное число.

За прошедшие годы для представления символов в памяти компьютера были разработаны различные схемы кодирования. Исторически самой важной из этих схем кодирования является схема кодирования ASCII (American Standard Code for Information Interchange – американский стандартный код обмена информацией). Аббревиатура ASCII произносится «аски».

ASCII представляет собой набор из 128 цифровых кодов, которые обозначают английские буквы, различные знаки препинания и другие символы. Например, код ASCII для прописной английской буквы A (латинской) равняется 65. Когда на компьютерной клавиатуре вы набираете букву A в верхнем регистре, в памяти сохраняется число 65 (как двоичное число, разумеется). Это показано на рис. 1.10.

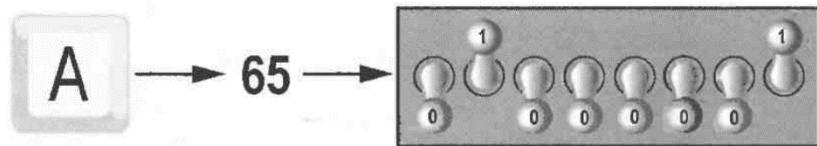


Рисунок 1.10 – Буква А хранится в памяти как 65_{10} или 01000001_2 .

Полная таблица символов ASCII выглядит следующим образом:

Код	Символ	Код	Символ	Код	Символ	Код	Символ	Код	Символ
0	NUL	26	SUB	52	4	78	N	104	h
1	SOH	27	ESC	53	5	79	O	105	i
2	STX	28	FS	54	6	80	P	106	j
3	ETX	29	GS	55	7	81	Q	107	k
4	EOT	30	RS	56	8	82	R	108	l
5	ENQ	31	US	57	9	83	S	109	m
6	ACK	32	(Пробел)	58	:	84	T	110	n
7	BEL	33	!	59	;	85	U	111	o
8	BS	34	"	60	<	86	V	112	p
9	HT	35	#	61	=	87	W	113	q
10	LF	36	\$	62	>	88	X	114	r
11	VT	37	%	63	?	89	Y	115	s
12	FF	38	&	64	@	90	Z	116	t
13	CR	39	'	65	A	91	[117	u
14	SO	40	(66	B	92	\	118	v
15	SI	41)	67	C	93]	119	w
16	DLE	42	*	68	D	94	^	120	x
17	DC1	43	+	69	E	95	_	121	y
18	DC2	44	,	70	F	96	.	122	z
19	DC3	45	-	71	G	97	a	123	{
20	DC4	46	.	72	H	98	b	124	
21	NAK	47	/	73	I	99	c	125	}
22	SYN	48	0	74	J	100	d	126	~
23	ETB	49	1	75	K	101	e	127	DEL
24	CAN	50	2	76	L	102	f		
25	EM	51	3	77	M	103	g		

Итак, вы познакомились с числами и их хранением в памяти. Во время чтения вы, возможно, подумали, что двоичная система исчисления может использоваться для представления только целых чисел, начиная с 0. Представить отрицательные и вещественные числа (такие как 3.14159) при помощи рассмотренного нами простого двоичного метода нумерации невозможно.

Однако компьютеры способны хранить в памяти и отрицательные, и вещественные числа, но для этого помимо двоичной системы исчисления в них используются специальные схемы кодирования. Отрицательные числа кодируются с использованием метода, который называется дополнением до двух, а вещественные числа кодируются в форме записи с плавающей точкой. От вас не требуется знать, как эти схемы кодирования работают, за исключением того, что они используются для преобразования отрицательных и вещественных чисел в двоичный формат.

Компьютеры принято называть цифровыми устройствами. Термин «цифровой» применяется для описания всего, что использует двоичные числа. Цифровые данные – это данные, которые хранятся в двоичном формате, цифровое устройство – любое устройство, которое работает с двоичными данными. Мы уже рассмотрели приемы хранения чисел и символов в двоичном формате, но компьютеры также работают со многими другими типами цифровых данных.

Например, возьмем снимки, которые вы делаете своей цифровой фотокамерой. Эти изображения состоят из крошечных точек цвета, которые называются пикселями. (термин «пикセル» образован от слов picture element – элемент изображения.) Каждый пиксель в изображении преобразуется в числовой код, который представляет цвет пикселя. Числовой код хранится в оперативной памяти как двоичное число.

1.3. Выполнение программ

ЦП компьютера может понимать только те инструкции, которые написаны на машинном языке. Поскольку людям очень сложно писать какие-либо программы на машинном языке, были придуманы другие языки программирования.

ЦП – самый важный компонент в компьютере, потому что это та его часть, которая выполняет программы. Иногда ЦП называют «мозгом» компьютера и даже используют эпитет «умный» (smart). Хотя эти метафоры встречаются очень часто, следует понимать, что ЦП не является мозгом и его нельзя назвать умным. ЦП – это электронное устройство, которое предназначено для выполнения конкретных вещей (жестко заданных алгоритмически).

В частности, ЦП служит для выполнения таких операций, как:

- ◆ чтение порции данных из оперативной памяти;
- ◆ сложение двух чисел;
- ◆ вычитание одного числа из другого;

- ◆ умножение двух чисел;
- ◆ деление одного числа на другое число;
- ◆ перемещение порции данных из одной ячейки оперативной памяти в другую;
- ◆ определение, равно ли одно значение другому значению.

Как видно из этого списка, ЦП выполняет простые операции с порциями данных. Однако ЦП ничего не делает самостоятельно. Ему нужно сообщить, что именно надо сделать, и это является целью программы. Программа – это не более чем список инструкций, которые заставляют ЦП выполнять операции.

Каждая инструкция в программе – это команда, которая поручает ЦП выполнить конкретную операцию. Вот пример инструкции, которая может появиться в программе:

10110000

Для нас это всего лишь вереница нулей и единиц. Для ЦП она является инструкцией с указанием выполнить какую-то операцию и записывается в виде нулей и единиц, потому что ЦП понимают только те инструкции, которые написаны на машинном языке, а инструкции машинного языка всегда имеют двоичную структуру.

Инструкция на машинном языке существует для каждой операции, которую ЦП способен выполнить. Весь перечень инструкций, которые ЦП может исполнять, называется набором инструкции ЦП.

Сегодня имеется несколько микропроцессорных компаний, которые изготавливают центральные процессоры. Самые известные – Intel, AMD и Motorola.

Каждый бренд микропроцессора имеет собственную уникальную систему инструкций, которая, как правило, понятна микропроцессорам только того же бренда. Например, микропроцессоры Intel понимают одинаковые инструкции, но они не понимают инструкции для микропроцессоров Motorola.

Ранее показанная инструкция на машинном языке является примером всего одной инструкции. Однако, для того чтобы компьютер делал что-то содержательное, ему требуется гораздо больше инструкций. Поскольку операции, которые ЦП умеет выполнять, являются элементарными, содержательная задача может быть выполнена, только если ЦП выполняет много операций. Например, если вы хотите, чтобы ваш компьютер вычислил сумму процентного дохода, которую вы получите на своем сберегательном счете в этом году, то ЦП должен будет исполнить большое количество инструкций в надлежащей последовательности.

Вполне обычной является ситуация, когда программы содержат тысячи или даже миллионы инструкций на машинном языке.

Программы обычно хранятся на вторичном устройстве хранения, таком как жесткий диск.

Когда вы устанавливаете программу на свой компьютер, она обычно скачивается с веб-сайта или устанавливается из интернет-магазина приложений.

Хотя программа размещается на вторичном устройстве хранения, таком как жесткий диск, тем не менее при каждом ее исполнении центральным процессором она должна копироваться в основную память, или ОЗУ. Например, предположим, что на диске вашего компьютера имеется программа обработки текста. Для ее исполнения вы делаете двойной щелчок мышью по значку программы. Это приводит к тому, что программа копируется с диска в основную память. Затем ЦП компьютера исполняет копию программы, которая находится в основной памяти (рис. 1.11)



Рисунок 1.11 – Программа копируется в основную память и затем исполняется.

Когда ЦП исполняет инструкции в программе, он участвует в процессе, который называется циклом выборки-декодирования-исполнения. Этот цикл состоит из трех шагов и повторяется для каждой инструкции в программе. Эти шаги следующие (рис. 1.12):

1. Выборка. Программа представляет собой длинную последовательность инструкций на машинном языке. Первый шаг цикла состоит в выборке, или чтении, следующей инструкции из памяти и ее доставки в ЦП.

2. Декодирование. Инструкция на машинном языке представляет собой двоичное число, обозначающее команду, которая даст указание ЦП выполнить операцию. На этом шаге ЦП декодирует инструкцию, которая только что была выбрана из оперативной памяти, чтобы определить, какую операцию он должен выполнить.

3. Исполнение. Последний шаг в цикле состоит в исполнении, или осуществлении, операции.

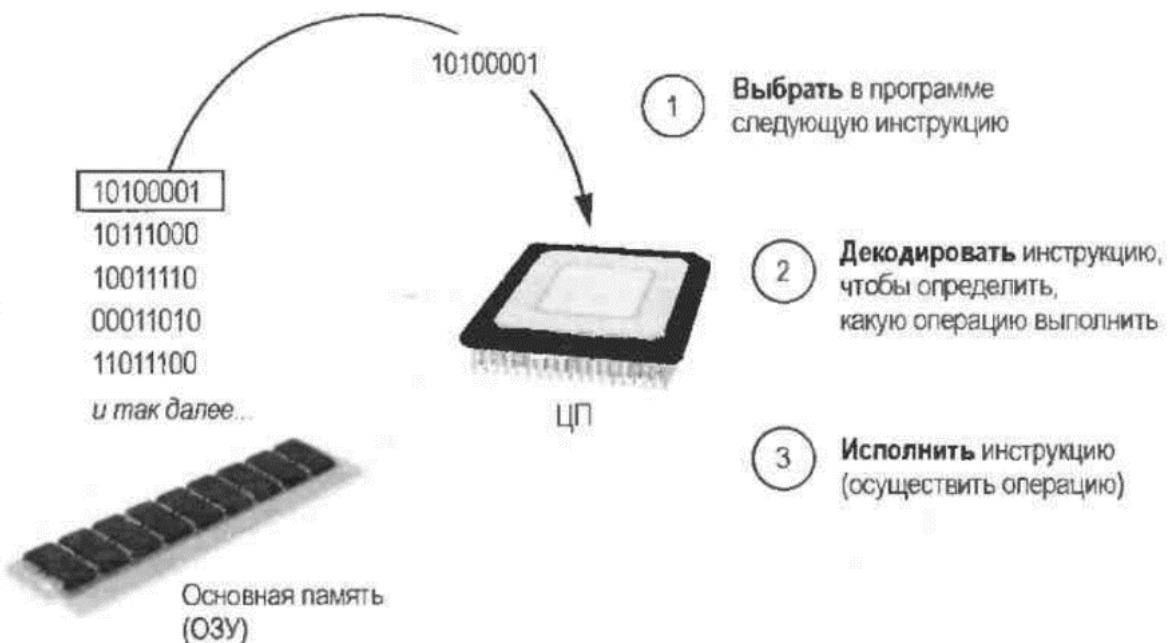


Рисунок 1.12 – Цикл выборки-декодирования-исполнения.

1.4. Компиляторы и интерпретаторы. Языки программирования

Компьютеры могут исполнять только те программы, которые написаны на машинном языке.

Как упоминалось ранее, программа может иметь тысячи или даже миллионы двоичных инструкций, и написание такой программы было бы очень утомительной и трудоемкой работой. Программировать на машинном языке будет тоже очень трудно, потому что если разместить 0 или 1 в неправильном месте, это вызовет ошибку.

Хотя ЦП компьютера понимает только машинный язык, люди практически неспособны писать программы на нем. По этой причине с первых дней вычислительных систем в качестве альтернативы машинному языку был создан язык ассемблера (от англ. assembly – сборка, монтаж, компоновка). Вместо двоичных чисел, которые соответствуют инструкциям, в языке ассемблера применяются короткие слова, которые называются мнемониками, или мнемокодами. Например, на языке ассемблера мнемоника add, как правило, означает сложить числа, mul – умножить числа, mov – переместить значение в ячейку оперативной памяти.

Когда для написания программы программист использует язык ассемблера, вместо двоичных чисел он имеет возможность записывать короткие мнемоники.

Существует много разных версий языка ассемблера. Ранее было упомянуто, что каждый бренд ЦП имеет собственную систему инструкций машинного языка. Как правило, каждый бренд ЦП также имеет свой язык ассемблера.

Вместе с тем программы на языке ассемблера ЦП исполнять не может. ЦП понимает только машинный язык, поэтому для перевода программы, написанной на языке ассемблера, в программу на машинном языке применяется специальная программа, которая называется ассемблером. Этот процесс показан на рис. 1.13. Программа на машинном языке, которая создается ассемблером, затем может быть исполнена центральным процессором.

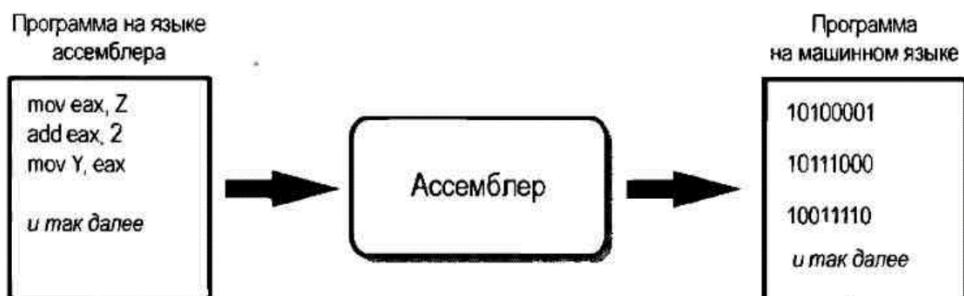


Рисунок 1.13 – Ассемблер переводит программу на языке ассемблера в программу на машинном языке.

Хотя язык ассемблера делает ненужным написание двоичных инструкций на машинном языке, он имеет свои сложности. Язык ассемблера прежде всего является непосредственной заменой машинного языка и подобно машинному языку он обязывает вас к тому, чтобы вы хорошо разбирались в ЦП. Язык ассемблера также требует, чтобы вы писали большое количество инструкций даже для самой простой программы. Поскольку язык ассемблера по своей природе непосредственно связан с машинным языком, он называется низкоуровневым языком.

В 1950-х годах появилось новое поколение языков программирования, которые называются высокоуровневыми языками. Высокоуровневый язык позволяет создавать мощные и сложные программы, не требуя знаний особенностей работы ЦП, и написания большого количества низкоуровневых инструкций. Кроме того, в большинстве высокоуровневых языков используются легко понимаемые слова. Например, если бы программист использовал COBOL (который был одним из ранних высокоуровневых языков, созданных в 1950-х годах), то, к примеру, для отображения на мониторе сообщения "Привет, мир!" он написал бы следующую инструкцию:

```
DISPLAY "Привет, мир!"
```

Python – это современный, высокоуровневый язык программирования, который будет изучаться в данном курсе. В Python сообщение «Привет, мир!» выводится на экран при помощи такой инструкции:

```
print('Привет, мир!')
```

Выполнение того же самого на языке ассемблера потребовало бы нескольких инструкций и глубокого знания того, как ЦП взаимодействует с компьютерным устройством вывода.

Как видно из этого примера, высокоуровневые языки позволяют программистам сосредоточиваться на задачах, которые они хотят выполнять при помощи своих программ, а не на деталях того, как ЦП будет исполнять эти программы.

Начиная с 1950-х годов были созданы тысячи высокоуровневых языков. В табл. 1.1 перечислены некоторые из наиболее известных языков.

Таблица 1.1 – Языки программирования

Язык	Описание
Ada	Был создан в 1970-х годах прежде всего для приложений, использовавшихся Министерством обороны США. Язык назван в честь графини Ады Лавлейс, влиятельной исторической фигуры в области вычислительных систем
BASIC	Универсальная система символьического кодирования для начинающих (Beginners All-purpose Symbolic Instruction Code) является языком общего назначения, который был первоначально разработан в начале 1960-х годов, как достаточно простой в изучении начинающими программистами. Сегодня существует множество разных версий языка BASIC

FORTRAN	Транслятор формул (FORmula TRANSlator) был первым высокоуровневым языком программирования. Он был разработан в 1950-х годах для выполнения сложных математических вычислений
COBOL	Универсальный язык для коммерческих задач (Common Business-Oriented Language) был создан в 1950-х гг. для бизнес-приложений
Pascal	Создан в 1970 г. и первоначально был предназначен для обучения программированию. Этот язык был назван в честь математика, физика и философа Блеза Паскаля
C и C++	C и C++ (произносится как "си плюс плюс") являются мощными языками общего назначения, разработанными в компании Bell Laboratories. Язык C был создан в 1972 г., язык C++ был выпущен в 1983 г.
C#	C# произносится "си шарп". Этот язык был создан компанией Microsoft примерно в 2000 г. для разработки приложений на основе платформы Microsoft .NET
Язык	Описание
Java	Был создан компанией Sun Microsystems в начале 1990-х годов. Он используется для разработки программ, которые работают на одиночном компьютере либо через Интернет с веб-сервера
JavaScript	Был создан в 1990-х годах и используется на веб-страницах. Несмотря на его название JavaScript с Java никак не связан
Python	Используемый в этой книге язык Python является языком общего назначения, который был создан в начале 1990-х годов. Он стал популярным в коммерческих и научных приложениях
Ruby	Ruby является языком общего назначения, который был создан в 1990-х годах. Он становится все более популярным языком для создания программ, которые работают на веб-серверах
Visual Basic	Широко известный как VB, является языком программирования Microsoft и средой разработки программного обеспечения, позволяет программистам быстро создавать Windows-ориентированные приложения. VB был создан в начале 1990-х годов

Поскольку ЦП понимает инструкции только на машинном языке, программы, написанные на высокоуровневом языке, должны быть переведены, или транслированы, на машинный язык. В зависимости от языка, на котором была написана программа, программист для выполнения трансляции использует компилятор либо интерпретатор.

Компилятор – это программа, которая транслирует программу на высокоуровневом языке в отдельную программу на машинном языке. Программа на машинном языке затем может быть выполнена в любое время, когда потребуется (рис. 1.14). Обратите внимание, что компиляция и исполнение – это два разных процесса.

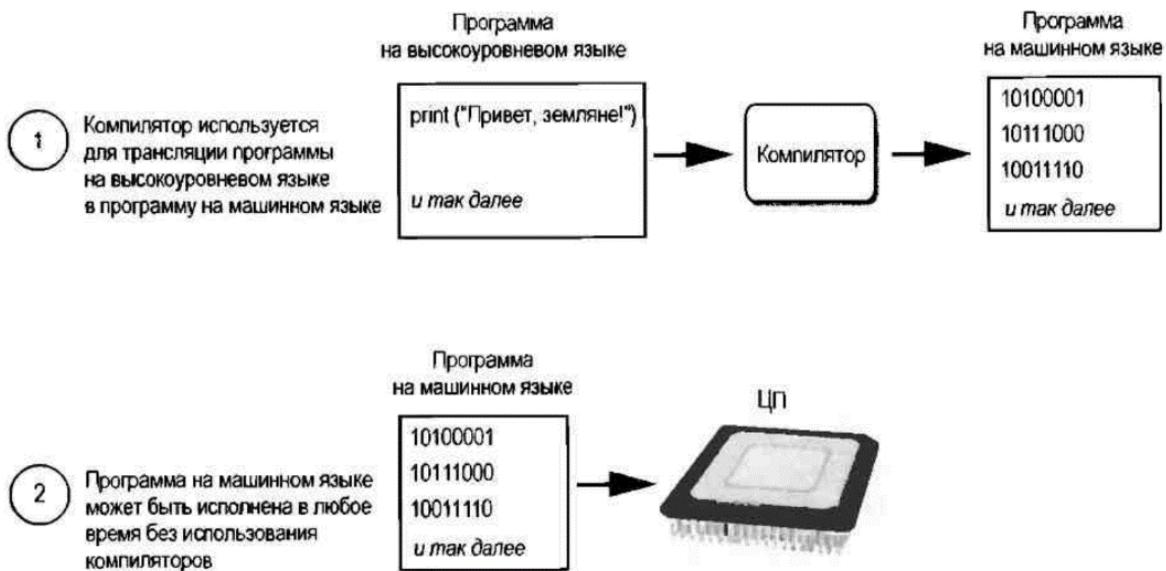


Рисунок 1.14 – Компиляция высокоуровневой программы и ее исполнение.

В языке Python используется интерпретатор. Это программа, которая одновременно транслирует и исполняет инструкции в программе, написанной на высокоуровневом языке. По мере того, как интерпретатор читает каждую отдельную инструкцию в программе, он ее преобразовывает в инструкции на машинном языке и затем немедленно их исполняет. Этот процесс повторяется для каждой инструкции в программе (рис. 1.15). Поскольку интерпретаторы объединяют процессы трансляции и исполнения, как правило, отдельные программы на машинном языке ими не создаются.

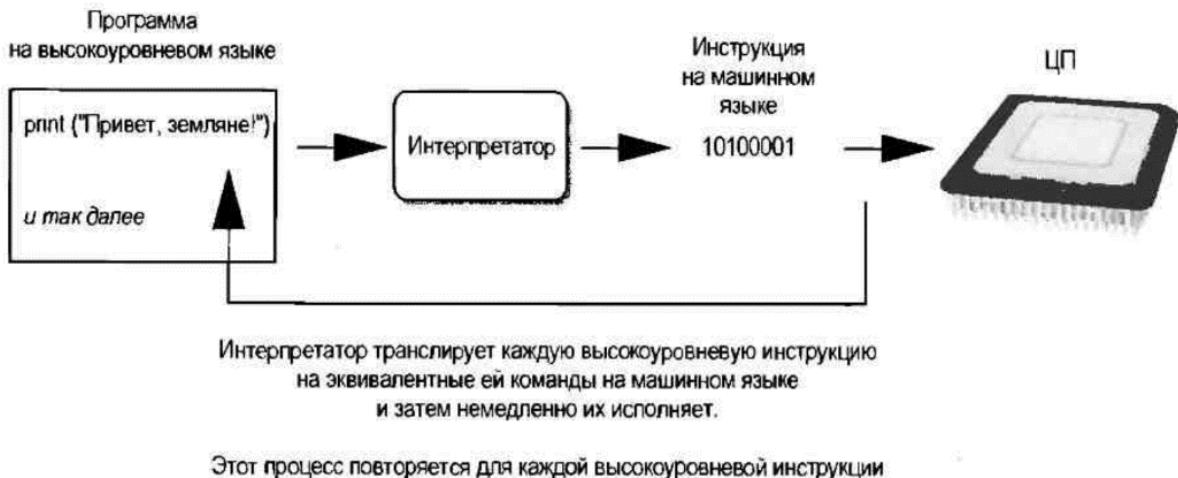


Рисунок 1.15 – Исполнение высокоуровневой программы интерпретатором

Инструкции, которые программист пишет на высокоуровневом языке, называются исходным кодом, программным кодом или просто кодом. Как

правило, программист набирает код программы в текстовом редакторе, затем сохраняет его в файле на диске компьютера. Далее программист использует компилятор для трансляции кода в программу на машинном языке либо интерпретатор для трансляции и исполнения кода. Однако, если программный код содержит синтаксическую ошибку, он не может быть транслирован. Синтаксическая ошибка – это неточность в программе, такая как ключевое слово с опечаткой, недостающий знак препинания или неправильно использованный оператор. Когда это происходит, компилятор или интерпретатор выводит на экран сообщение об ошибке, говорящее, что программа содержит синтаксическую ошибку. Программист исправляет ошибку, затем пробует транслировать программу еще раз.

Вопросы для самопроверки по теме 1

1. Что такое программа?
2. Что такое аппаратное обеспечение?
3. Перечислите пять главных компонентов компьютерной системы.
4. Какая часть компьютера исполняет программы фактически?
5. Какая часть компьютера служит рабочей областью для хранения программы и ее данных, пока программа работает?
6. Какая часть компьютера содержит данные в течение долгого времени, даже когда к компьютеру не подведено электропитание?
7. Какая часть компьютера собирает данные от людей и от различных устройств?
8. Какая часть компьютера форматирует и предоставляет данные людям и подключенными к нему устройствам?
9. Какой фундаментальный набор программ управляет внутренними операциями аппаратного обеспечения компьютера?
10. Как называется программа, которая выполняет специализированную задачу, в частности, такие программы, как вирусный сканер, программа сжатия файлов или программа резервного копирования данных?
11. К какой категории программного обеспечения принадлежат программы обработки текста, программы по работе электронными таблицами, почтовые программы, веб-браузеры и компьютерные игры?
12. Какого объема памяти достаточно для хранения буквы алфавита или небольшого числа?
13. В какой системе исчисления все числовые значения записываются как последовательности нулей и единиц?
14. Какова задача схемы кодирования ASCII?
15. Какая схема кодирования является достаточно широкой, чтобы включать символы многих языков мира?

16. Что означают термины «цифровые данные» и «цифровое устройство»?

17. ЦП понимает инструкции, которые написаны только на одном языке. Как называется этот язык?

18. Как называется тип памяти, в которую программа должна копироваться при каждом ее исполнении центральным процессором?

19. Как называется процесс, в котором участвует ЦП, когда он исполняет инструкции в программе?

20. Что такое язык ассемблера?

21. Какой язык программирования позволяет создавать мощные и сложные программы, не разбираясь в том, как работает ЦП?

22. Каждый язык имеет набор правил, который должен строго соблюдаться во время написания программы. Как называется этот набор правил?

23. Как называется программа, которая транслирует программу, написанную на высокоуровневом языке, в отдельную программу на машинном языке?

24. Как называется программа, которая одновременно транслирует и исполняет инструкции программы на высокоуровневом языке?

25. Причинами какого типа ошибок обычно является ключевое слово с опечаткой, недостающий знак препинания или неправильно использованный оператор?

Лекция 2. Принципы разработки программ

План лекции

1. Средства разработки Python.
2. Структура скрипта Python.
3. Цикл проектирования программы.
4. Псевдокод. Блок-схема.
5. Обработка и вывод данных.
6. Переменные. Ввод данных
7. Вычислительные задачи.

2.1. Средства разработки Python

Интерпретатор Python исполняет программы Python, которые хранятся в файлах, либо оперативно исполняет инструкции Python, которые набираются на клавиатуре в интерактивном режиме. Python поставляется вместе с программой под названием IDLE, которая упрощает процесс написания, исполнения и тестирования программ.

Прежде чем продолжить изучение данного курса необходимо убедиться в наличии средства разработки Python/ Если вы работаете в компьютерном классе, то это уже, по-видимому, было сделано.

Необходимо понимать и уметь реализовывать различные подходы к написанию и выполнению кода Python. Выделим следующие подходы:

1. Применение интерпретатора Python, установленного локально на компьютере. Наиболее простой способ установить интерпретатор Python – это использование программного обеспечения Anaconda. Скачать и установить можно по адресу: <https://www.anaconda.com>

Часто при наличии локального интерпретатора используют редакторы для написания программ: IDLE, Sublime Text Editor или интегрированные среды разработки Visual Studio, PyCharm.

2. Применение онлайн редактора. В настоящее время существует большое разнообразие Интернет-ресурсов, предоставляющих подобный сервис. При таком подходе интерпретатор Python отсутствует на локальном компьютере пользователя, программист вводит программные инструкции в поля в браузере (используя его как редактор), а интерпретатор установлен на удаленном компьютере-сервере. Примеры онлайн-интерпретаторов Python:

Online Python: <https://www.online-python.com>

Programiz: <https://www.programiz.com/python-programming/online-compiler/>

OnlineGDB: https://www.onlinegdb.com/online_python_compiler

W3School: https://www.w3schools.com/python/python_compiler.asp

Существует большое количество других онлайн-интерпретаторов.

3. Применение онлайн среды разработки Google Colaboratory. Данный подход существенно расширяет подход 2, также не требует локальной установки интерпретатора Python. Доступ к облачному сервису GoogleColab производится по ссылке: <https://colab.research.google.com>

В рамках данного курса слушателям следует разобраться и поработать с различными подходами к разработке и исполнению программ Python. Практические занятия будут ориентироваться на GoogleColab.

2.2. Структура скрипта Python.

Часто последовательность команд – программу на языке Python, – называют скриптом. Поэтому условимся, что скрипт – это набор команд на языке Python, сохраненный в текстовом файле.

Например, предположим, что вы хотите написать программу Python, которая выводит на экран приведенные далее три строки текста:

Мигнуть

Моргнуть

Кивнуть.

Для написания программы следует создать файл в простом текстовом редакторе, таком как Блокнот (который установлен на всех компьютерах с Windows), содержащий следующие инструкции:

```
print('Мигнуть')
print('Моргнуть')
print('Кивнуть.')
```

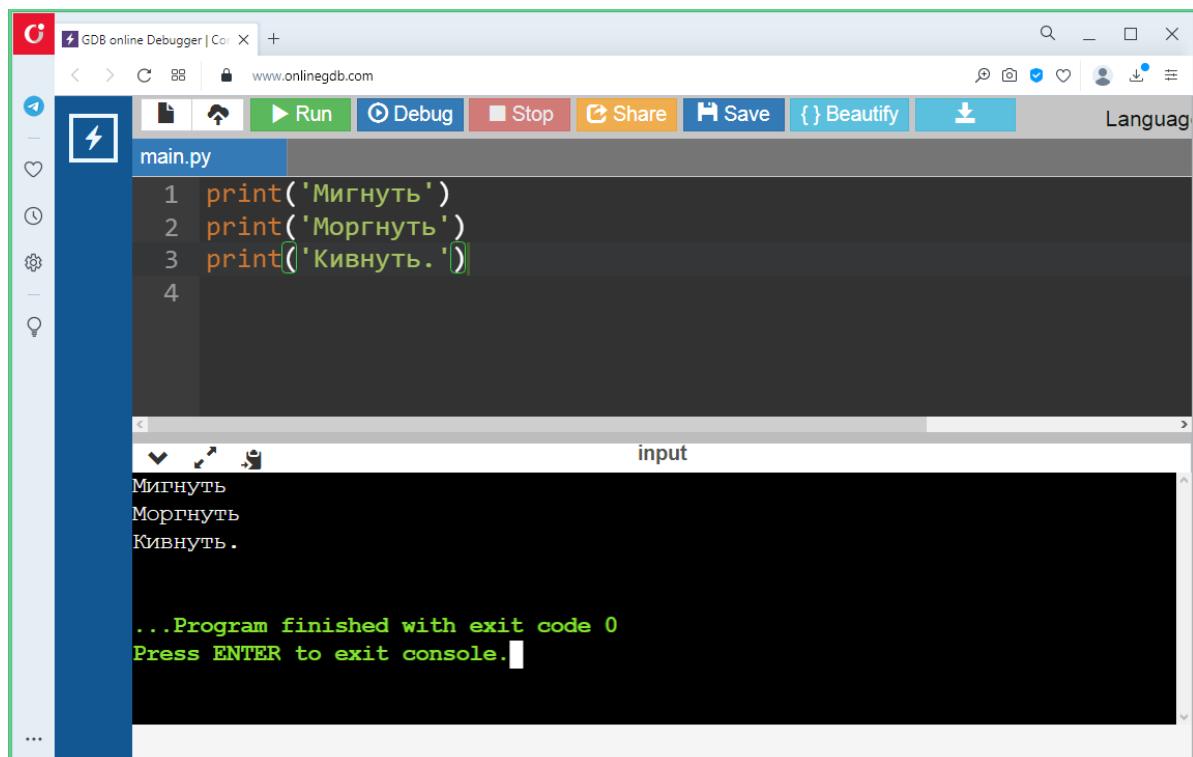
Для создания программы Python можно использовать текстовой процессор, но вы должны убедиться, что сохраняете программу как файл с обычным текстом. В противном случае интерпретатор Python не сможет прочитать его содержимое.

При сохранении программы Python ей следует дать имя с расширением `.py`, которое идентифицирует ее как программу Python. Например, приведенную выше программу можно сохранить под именем `test.py`. Для того чтобы выполнить программу, следует перейти в каталог, в котором сохранен файл, и в командной оболочке операционной системы набрать команду:

```
python test.py
```

Эта команда запустит интерпретатор Python в сценарном режиме, в результате чего он исполнит инструкции в файле `test.py`. Когда программа закончит исполняться, интерпретатор Python прекратит свою работу.

Эту же программу можно набрать (скопировать) в онлайн редактор (например OnlineGDB) и выполнить скрипт, нажав кнопку  `▶ Run`.



The screenshot shows a web-based debugger interface for Python scripts. The main window displays the code in a file named `main.py`:

```
1 print('Мигнуть')
2 print('Моргнуть')
3 print('Кивнуть.')
4
```

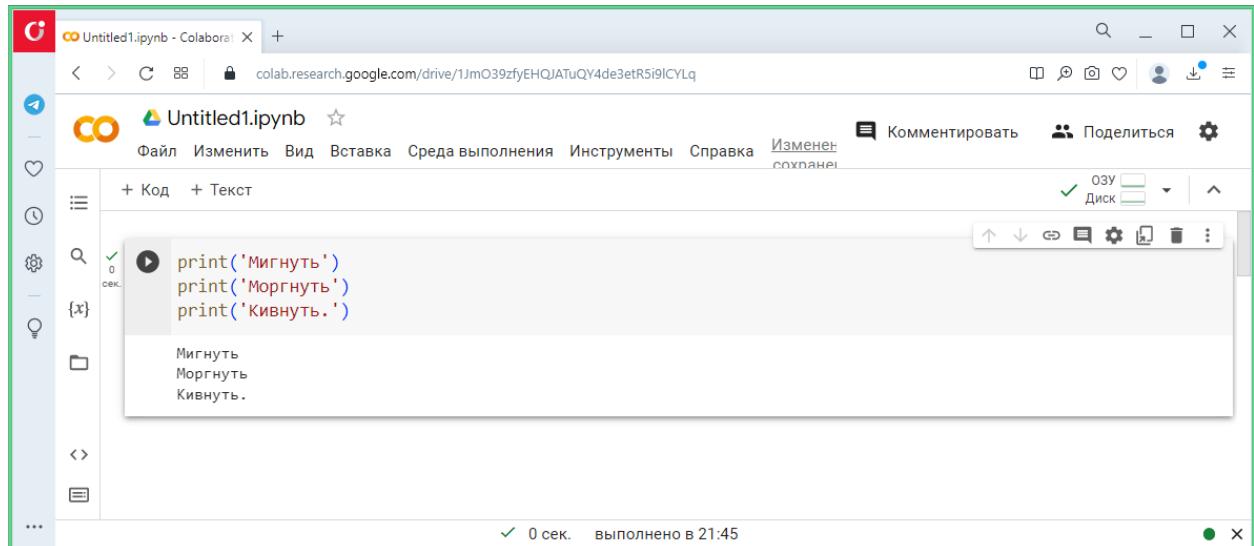
Below the code, the output window shows the results of the `print` statements:

```
Мигнуть
Моргнуть
Кивнуть.
```

At the bottom of the output window, the message "...Program finished with exit code 0" is displayed, followed by "Press ENTER to exit console."

Рисунок 2.1 – Работа со скриптом в online-интерпретаторе

Можно также реализовать выполнение скрипта в GoogleColab



The screenshot shows a Google Colab notebook titled `Untitled1.ipynb`. The code cell contains the same Python script as in Figure 2.1:

```
print('Мигнуть')
print('Моргнуть')
print('Кивнуть.')
```

The output of the cell is shown below the code:

```
Мигнуть
Моргнуть
Кивнуть.
```

At the bottom of the notebook interface, it says "0 сек. выполнено в 21:45".

Рисунок 2.2 – Работа со скриптом в GoogleColab

2.3. Цикл проектирования программы

Для создания программ разработчики используют преимущественно высокоуровневые языки, такие как Python. Однако создание программы отнюдь не ограничивается написанием кода. Процесс создания правильно

работающей программы, как правило, подразумевает пять фаз, показанных на рис. 2.3. Весь этот процесс называется циклом разработки программы.



Рисунок 2.3 – Цикл разработки программы

Рассмотрим каждый этап данного цикла подробнее.

1. Спроектировать программу. Все профессиональные программисты вам скажут, что до того, как рабочий код программы будет написан, программа должна быть тщательно спроектирована. Когда программисты начинают новый проект, они никогда не бросаются к написанию кода с места в карьер. Они начинают с того, что создают проект программы. Существует несколько способов проектирования программы, и позже в этом разделе мы рассмотрим отдельные методы, которые можно применять для разработки программ Python.

2. Написать код. После создания проекта программист начинает записывать код на высокоуровневом языке, в частности на Python. Каждый язык имеет свои синтаксические правила, которые должны соблюдаться при написании программы. Эти правила языка диктуют, каким образом использовать такие элементы, как ключевые слова, операторы и знаки препинания. Синтаксическая ошибка происходит в случае, если программист нарушает какое-либо из этих правил.

3. Исправить синтаксические ошибки. Если программа содержит синтаксическую ошибку или даже простую неточность, такую как ключевое слово с опечаткой, то компилятор или интерпретатор выведет на экран сообщение об ошибке с ее описанием. Практически любой программный код содержит синтаксические ошибки, когда он написан впервые, поэтому, как правило, программист будет тратить некоторое время на их исправление. После того как все синтаксические ошибки и простые неточности набора

исходного кода исправлены, программа может быть скомпилирована и транслирована в программу на машинном языке (либо, в зависимости от используемого языка, исполнена интерпретатором).

4. Протестировать программу. Когда программный код находится в исполнимой форме, его тестируют с целью определения каких-либо логических ошибок. Логическая ошибка – это неточность, которая не мешает выполнению программы, но приводит к неправильным результатам. (математические неточности являются типичными причинами логических ошибок).

5. Исправить логические ошибки. Если программа приводит к неправильным результатам, программист выполняет отладку кода – отыскивает в программе логические ошибки и исправляет их. Иногда во время этого процесса программист обнаруживает, что оригинальный проект программы должен быть изменен. Тогда разработка программы вновь начинается и продолжается до тех пор, пока все ошибки не будут обнаружены и устранены.

2.4. Псевдокод. Блок-схема.

Поскольку неточности, т. е. слова с опечатками и пропущенные знаки препинания, могут вызывать синтаксические ошибки, программисты должны быть внимательны к таким мелким деталям при написании кода. По этой причине, прежде чем написать программу в рабочем коде языка программирования, в частности на Python, программисты считают полезным написать программу в псевдокоде (т. е. с пропуском несущественных подробностей).

Слово "псевдо" означает "фикция, подделка", поэтому псевдокод – это фиктивный код. Это неформальный язык, который не имеет каких-либо синтаксических правил и не предназначен для компиляции или исполнения. Вместо этого для создания моделей, или макетов, программ разработчики используют псевдокод. Поскольку при написании псевдокода программистам

не приходится беспокоиться о синтаксических ошибках, они могут спокойно сосредоточить все свое внимание на проектировании программы. После того как на основе псевдокода создан отвечающий требованиям проект, псевдокод может быть переведен непосредственно в рабочий код.

Предположим, вам поручили написать программу, которая рассчитывает и показывает заработную плату до налоговых и прочих удержаний для сотрудника с почасовой ставкой оплаты труда. Вот шаги, которые вы бы проделали:

1. Получить количество отработанных часов.
2. Получить почасовую ставку оплаты труда.
3. Умножить число отработанных часов на почасовую ставку оплаты труда.
4. Показать результат вычисления, выполненного на шаге 3.

Разумеется, этот алгоритм совсем не готов к тому, чтобы его можно было исполнить на компьютере. Шаги, перечисленные в этом списке, сначала должны быть переведены в программный код. Для достижения этой цели программисты широко применяют два инструмента: псевдокод и блок-схемы.

Вот пример того, как можно написать псевдокод для программы вычисления зарплаты, которую мы рассмотрели ранее:

Ввести отработанные часы.

Ввести почасовую ставку оплаты труда.

Рассчитать заработную плату до удержаний, как произведение отработанных часов и ставки оплаты труда.

Показать заработную плату.

Каждая инструкция в псевдокоде представляет операцию, которая может быть выполнена на языке Python. Например, Python может прочитать входные данные, набираемые на клавиатуре, выполнить математические расчеты и показать сообщения на экране.

Блок-схемы являются еще одним инструментом, который программисты используют для проектирования программ. Блок-схема — это диаграмма,

которая графически изображает шаги в программе. Блок-схема для программы расчета заработной платы представлена на рис. 2.4.



Рисунок 2.4 – Блок-схема программы расчета заработной платы

Обратите внимание, что в блок-схеме имеется три типа символов: овалы, параллелограммы и прямоугольники. Каждый из этих символов представляет шаг в программе, как описано далее.

- ◆ Овалы, которые появляются вверху и внизу блок-схемы, называются терминальными символами. Терминальный символ Начало отмечает начальную точку программы, терминальный символ Конец – ее конечную точку.
- ◆ Параллелограммы используются в качестве входных и выходных символов. Они обозначают шаги, в которых программа считывает данные на входе (т. е. входные данные) или показывает итоговые данные на выходе (т. е. выходные данные).

◆ Прямоугольники используются в качестве обрабатывающих символов. Они обозначают шаги, в которых программа выполняет некую обработку данных, такую как математическое вычисление.

Символы соединены стрелками, которые представляют «поток» вычислений программы. Для того чтобы пройти символы в надлежащем порядке, нужно начать с терминального символа *Начало* и следовать вдоль стрелок, пока не будет достигнут терминальный символ *Конец*.

2.5. Обработка и вывод данных.

Входные данные – это данные, которые программа получает на входе. При получении данных программа обычно их обрабатывает путем выполнения над ними некой операции. Выходные данные, или итоговый результат операции, выводятся из программы.

Компьютерные программы, как правило, выполняют приведенный ниже трехшаговый процесс:

1. Получить входные данные (ввести данные).
2. Выполнить некую обработку входных данных.
3. Выдать выходные данные (вывести данные).

Входные данные – это любые данные, которые программа получает во время своего выполнения. Одной из типичных форм входных данных являются данные, вводимые с клавиатуры. После того как входные данные получены, обычно они подвергаются некой обработке, такой как математическое вычисление. Результаты этой обработки отправляются из программы в качестве выходных данных.

На рис. 2.5 показаны шаги в программе расчета заработной платы, которую мы рассмотрели ранее. Количество отработанных часов и почасовая ставка оплаты труда передаются в качестве входных данных. Программа обрабатывает эти данные путем умножения отработанных часов на почасовую ставку оплаты труда. Результаты расчетов выводятся в качестве выходных данных.

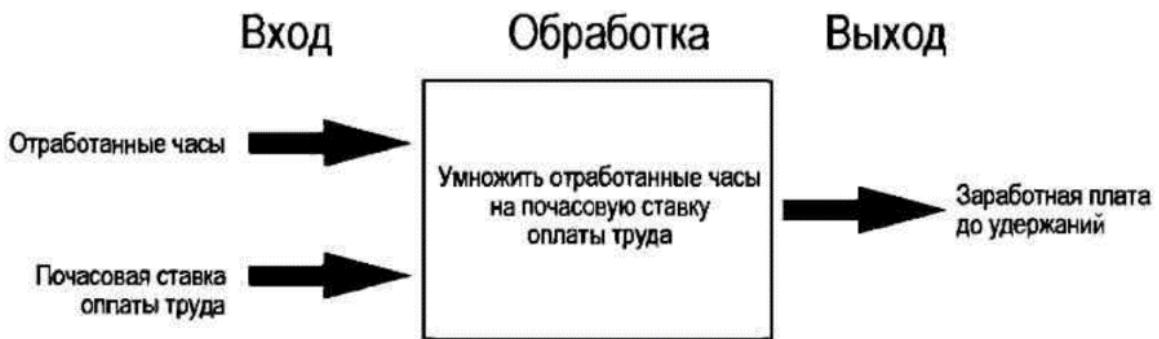


Рисунок 2.5 – Ввод, обработка и вывод программы расчета заработной платы

2.5.1. Вывод данных на экран при помощи функции print

Функция `print` используется для вывода на экран выходных данных в программе Python.

Функция – это фрагмент заранее написанного кода, который выполняет некую операцию. Python имеет многочисленные встроенные функции, которые выполняют различные операции. Возможно, самой фундаментальной встроенной функцией является функция печати `print`, которая показывает выходные данные на экране. Вот пример инструкции, которая исполняет функцию `print`:

```
print('Привет, мир!')
```

Когда программисты исполняют функцию, они говорят, что вызывают функцию. При вызове функции `print` набирается слово `print`, а затем пара круглых скобок. Внутри круглых скобок набирается аргумент, т. е. данные, которые требуется вывести на экран. В предыдущем примере аргументом является '`Привет, мир!`'. Отметим, что во время исполнения этой инструкции символы кавычек не выводятся. Символы кавычек просто задают начало и конец текста, который вы хотите показать.

Предположим, что ваш преподаватель поручает вам написать программу, которая выводит на мониторе имя и адрес. В представленной ниже программе представлен код с результатом, который она произведет при ее выполнении:

Листинг исходного кода Python:

```
print('Кейт Остен')
print('123 Фул Серкл Драйв')
print('Эшвиль, Северная Каролина 28899')
```

Вывод программы:

```
Кейт Остен
123 Фул Серкл Драйв
Эшвиль, Северная Каролина 28899
```

Важно понять, что инструкции в этой программе исполняются в том порядке, в котором они появляются в программе сверху вниз. При выполнении этой программы исполнится первая инструкция, вслед за ней вторая инструкция и затем третья.

Программы почти всегда работают с данными какого-то типа. Например, приведенная ранее программа использует три приведенные порции данных:

```
'Кейт Остен'
'123 Фул Серкл Драйв'
'Эшвиль, Северная Каролина 28899'
```

Эти порции данных представляют собой цепочки символов. В терминах программирования цепочка символов, которая используется в качестве данных, называется символьной последовательностью, или строковым значением, или просто строкой. Когда символьная последовательность появляется в рабочем коде программы, она называется строковым литералом. В программном коде Python строковые литералы должны быть заключены в знаки кавычек. Как отмечалось ранее, знаки кавычек просто отмечают, где строковые данные начинаются и заканчиваются.

В Python можно заключать строковые литералы в одинарные кавычки ('') ибо двойные кавычки (""). То есть следующая программа полностью идентична предыдущей:

```
print("Кейт Остен")
print("123 Фул Серкл Драйв")
print("Эшвиль, Северная Каролина 28899")
```

2.5.2. Комментарии

Комментарии – это описательные пояснения, которые документируют строки программы или ее разделы. Комментарии являются частью программы, но интерпретатор Python их игнорирует. Они предназначены для людей, которые, возможно, будут читать исходный код.

Комментарии – это короткие примечания, которые размещаются в разных частях программы и объясняют, как эти части программы работают. Несмотря на то, что комментарии являются критически важной частью программы, интерпретатор Python их игнорирует.

Комментарии адресованы любому человеку, который будет читать программный код, и не предназначены для компьютера.

В Python комментарий начинается с символа решетки `#`. Когда интерпретатор Python видит символ `#`, он игнорирует все, что находится между этим символом и концом строки кода.

Например, взгляните на следующую программу... Строки кода 1 и 2 – комментарии, которые объясняют цель программы.

```
# Эта программа показывает
# ФИО и адрес
print("Кейт Остен")
print("123 Фул Серкл Драйв")
print("Эшвиль, Северная Каролина 28899")
```

В своем коде программисты чаще всего используют концевые комментарии. Концевой комментарий – это комментарий, который появляется в конце строки кода. Он обычно объясняет инструкцию, которая расположена в этой строке. В следующей программе приведен пример, в котором каждая

строка кода заканчивается комментарием, кратко объясняющим, что эта строка кода делает.

```
print('Кейт Остен')                      # Показать полное имя.  
print('123 Фул Серкл Драйв')           # Показать адрес проживания.  
print('Эшвилль, Северная Каролина 28899') # Показать город и индекс.
```

2.6. Переменные. Ввод данных

Переменная – это имя, которое представляет место хранения в памяти компьютера.

Программы обычно хранят данные в оперативной памяти компьютера и выполняют операции с этими данными. Например, рассмотрим типичный опыт совершения покупок в онлайн-магазине: вы просматриваете веб-сайт и добавляете в корзину товары, которые хотите приобрести. По мере того как вы добавляете товары, данные об этих товарах сохраняются в памяти. Затем, когда вы нажимаете кнопку оформления заказа, выполняющаяся на компьютере веб-сайта программа вычисляет стоимость всех товаров, которые находятся в вашей корзине, с учетом стоимости доставки и итоговой суммы всех сборов. При выполнении этих расчетов программа сохраняет полученные результаты в памяти компьютера.

Программы используют переменные для хранения данных в памяти. Переменная – это имя, которое представляет значение в памяти компьютера. Например, в программе, вычисляющей налог с продаж на приобретаемые товары, для представления этого значения в памяти может использоваться имя переменной `tax` (налог). Тогда как в программе, которая вычисляет расстояние между двумя городами, для представления этого значения в памяти может использоваться имя переменной `distance` (расстояние). Когда переменная представляет значение в памяти компьютера, мы говорим, что переменная ссылается на это значение.

2.6.1. Создание переменных инструкцией присваивания

Инструкция присваивания используется для создания переменной, которая будет ссылаться на порцию данных. Вот пример инструкции присваивания:

```
age = 25
```

После исполнения этой инструкции будет создана переменная с именем `аде` (возраст), и она будет ссылаться на значение 25. Этот принцип показан на рис. 2.6: здесь число 25 следует рассматривать как значение, которое хранится где-то в оперативной памяти компьютера.



Рисунок 2.6 – Переменная `age` ссылается на значение 25

Стрелка, которая направлена от имени `аде` в сторону значения 25, говорит, что имя `аде` этой переменной ссылается на это значение.

Инструкция присваивания записывается в приведенном ниже общем формате:

```
переменная = выражение
```

Знак "равно" (`=`) называется оператором присваивания. В данном формате переменная – это имя переменной, а выражение – значение либо любая порция программного кода, которая в результате дает значение. После исполнения инструкции присваивания переменная, заданная слева от оператора `=`, будет ссылаться на значение, заданное справа от оператора `=`.

Далее, как показано ниже, можно применить функцию `print` для отображения значений, на которые эти переменные ссылаются:

Исходный код программы:

```
width = 10
length = 5
print (width)
print (length)
```

Вывод программы:

10**5**

Во время передачи в функцию print переменной в качестве аргумента не следует заключать имя переменной в кавычки. Для того чтобы продемонстрировать причину, взгляните на приведенный ниже исходный код:

Исходный код программы:

```
width = 10
length = 5
print (width)
print ('length')
```

Вывод программы:

```
10
length
```

Переменную нельзя использовать, пока ей не будет присвоено значение. Если попытаться выполнить операцию с переменной, например напечатать ее, до того, как ей будет присвоено значение, то произойдет ошибка.

Иногда ошибка может быть вызвана простой опечаткой при наборе. Одним из таких примеров является имя переменной с опечаткой:

```
temperature = 74.5      # Создать переменную
print(tempereture)      # Ошибка! Имя переменной с опечаткой
```

Этот код вызовет ошибку, потому что в Python имена переменных чувствительны к регистру символов:

```
temperature = 74.5      # Создать переменную
print(Temperature)     # Ошибка! Неоднообразное применение регистра
```

2.6.2. Правила именования переменных

Хотя разрешается придумывать переменным свои имена, необходимо соблюдать правила.

- ◆ В качестве имени переменной нельзя использовать одно из ключевых слов Python.
- ◆ Имя переменной не может содержать пробелы.
- ◆ Первый символ должен быть одной из букв от a до z, от A до Z либо символом подчеркивания _.
- ◆ После первого символа можно использовать буквы от a до z или от A до Z, цифры от 0 до 9 либо символы подчеркивания.
- ◆ Символы верхнего и нижнего регистров различаются. Это означает, что имя переменной itemsOrdered (ЗаказаноТоваров) не является тем же, что и itemsordered (заказанотоваров).

В дополнение к соблюдению этих правил также всегда следует выбирать имена переменных, которые дают представление о том, для чего они используются. Например, переменная для температуры может иметь имя temperature, а переменную для скорости автомобиля можно назвать speed. У вас может возникнуть желание давать переменным имена, типа x и b2, но такие имена не дают ключ к пониманию того, для чего переменная предназначена.

Поскольку имя переменной должно отражать ее назначение, программисты часто оказываются в ситуации, когда им приходится создавать имена из нескольких слов. Например, посмотрите на приведенные ниже имена переменных:

```
grosspay
payrate
hotdogssoldtoday
```

К сожалению, эти имена с трудом удаётся прочесть, потому что слова в них не отделены. Поскольку в именах переменных нельзя использовать

пробелы, нужно найти другой способ отделять слова в многословном имени переменной и делать его более удобочитаемым для человека.

Один из способов – использовать символ подчеркивания вместо пробела. Например, приведенные ниже имена переменных читаются проще, чем показанные ранее:

```
gross_pay
pay_rate
hot_dogs_sold_today
```

Этот стиль именования переменных популярен среди программистов на Python. Правда, есть и другие стили, такие как горбатый стиль написания имен переменных. Имена переменных в горбатом стиле записываются так:

- ◆ имя переменной начинается с букв в нижнем регистре;
- ◆ первый символ второго и последующих слов записывается в верхнем регистре.

Например, приведенные ниже имена переменных написаны в горбатом стиле:

```
grossPay
payRate
hotDogsSoldToday
```

В табл. 2.1 перечислено несколько примеров имен переменных и указано, какие из них допустимы в Python и какие нет.

Таблица 2.1 – Примеры имен переменных

Имя переменной	Допустимое или недопустимое
<code>units_per_day</code>	Допустимое
<code>dayOfWeek</code>	Допустимое
<code>3dGraph</code>	Недопустимое. Имена переменных не могут начинаться с цифры
<code>June1997</code>	Допустимое
<code>Mixture#3</code>	Недопустимое. В именах переменных могут использоваться только буквы, цифры или символы подчеркивания

2.6.3. Вывод нескольких значений при помощи функции print

Python позволяет выводить несколько значений одним вызовом функции `print`. Рассмотрим программу:

Исходный код программы:

```
room = 507  
print('Я нахожусь в комнате номер')  
print(room)
```

Вывод программы:

```
Я нахожусь в комнате номер  
507
```

На самом деле нет необходимости использовать два вызова функции `print`:

Исходный код программы:

```
room = 507  
print('Я нахожусь в комнате номер', room)
```

Вывод программы:

```
Я нахожусь в комнате номер 507
```

В строке 2 мы передали в функцию `print` два аргумента: первый аргумент – это строковый литерал '**Я нахожусь в комнате номер**', второй аргумент – переменная `room`. Во время исполнения функция `print` вывела значения этих аргументов в том порядке, в каком мы их передали функции. Обратите внимание, что функция `print` автоматически напечатала пробел, разделяющий значения. Когда в функцию `print` передаются многочисленные аргументы (через запятую), при их выводе они автоматически отделяются пробелом.

2.6.4. Повторное присваивание значений переменным

Переменные называются так потому, что во время работы программы они могут ссылаться на разные значения. Когда переменной присваивается значение, она будет ссылаться на это значение до тех пор, пока ей не будет присвоено другое значение. Например, в следующей программе инструкция в строке 3 создает переменную с именем roubles и присваивает ей значение 2.75 (верхняя часть рис. 2.7).

```
1 # Эта программа показывает повторное присвоение значения переменной.
2 # Присвоить значение переменной roubles.
3 roubles = 2.75
4 print('У меня на счете', roubles, 'рублей.')
5 # Повторно присвоить значение переменной roubles,
6 # чтобы она ссылалась на другое значение.
7 roubles = 99.95
8 print('А теперь там', roubles, 'рублей!')
```

У меня на счете 2.75 рублей.
А теперь там 99.95 рублей!

Остаток рублей после исполнения строки 3



Остаток рублей после исполнения строки 8

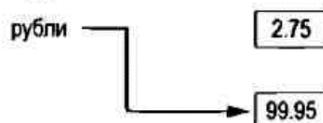


Рисунок 2.7 – Повторное присвоение значения переменной в программе

Затем инструкция в строке 7 присваивает переменной roubles другое значение – 99.95. В нижней части рис. 2.7 показано, как это изменяет переменную roubles. Старое значение 2.75 по-прежнему находится в памяти компьютера, но оно больше не может использоваться, потому что на него переменная не ссылается. Когда переменная больше не ссылается на значение в памяти, интерпретатор Python автоматически его удаляет из памяти посредством процедуры, которая называется сборщиком мусора.

2.6.5. Числовые типы данных и числовые литералы

Для хранения вещественных чисел (чисел с дробной частью) компьютеры используют прием, который отличается от хранения целых чисел. Мало того, что этот тип чисел хранится в памяти по-другому, но и аналогичные операции с ними тоже выполняются иначе.

Поскольку разные типы чисел хранятся и обрабатываются по-разному, в Python используются типы данных с целью классификации значений в оперативной памяти. Когда в оперативной памяти хранится целое число, оно классифицируется как **int**, а когда в памяти хранится вещественное число, оно классифицируется как **float**.

Интерпретатор Python определяет у числа тип данных. В нескольких приведенных ранее программах числовые данные записаны внутри программного кода. Например, в приведенной ниже инструкции записано число 503:

```
room = 503
```

Эта инструкция приводит к тому, что значение 503 сохраняется в оперативной памяти, и переменная `room` начинает ссылаться на это значение. В приведенной ниже инструкции записано число 2.75:

```
roubles = 2.75
```

Эта инструкция приводит к тому, что значение 2.75 сохраняется в оперативной памяти, и переменная `roubles` начинает ссылаться на это значение. Число, которое записано в коде программы, называется числовым литералом.

Когда интерпретатор Python считывает числовой литерал в коде программы, он определяет его тип данных согласно следующим правилам:

- ◆ числовой литерал, который записан в виде целого числа без десятичной точки, имеет целочисленный тип `int`, например 7,124 и -9;
- ◆ числовой литерал, который записан с десятичной точкой, имеет вещественный тип `float`, например 1.5, -3.14159 и 5.0.

Так, в приведенной далее инструкции значение 503 сохраняется в памяти как int:

```
room = 503
```

А другая инструкция приводит к тому, что значение 2.75 сохраняется в памяти как float:

```
roubles = 2.75
```

Когда значение сохраняется в оперативной памяти, очень важно понимать, о значении какого типа данных идет речь. Для установления типа переменной можно воспользоваться встроенной функцией **type**.

```
1 room = 211
2 roubles = 99.999
3 print(room, type(room))
4 print(roubles, type(roubles))
```

```
211 <class 'int'>
99.999 <class 'float'>
```

В дополнение к целочисленным типам данных int и вещественным типам данных float, Python имеет тип данных str, который используется для хранения в оперативной памяти строковых данных:

```
1 first_name='Evgeny'
2 last_name="Nikolaev"
3 print(first_name, type(first_name))
4 print(last_name, type(last_name))
```

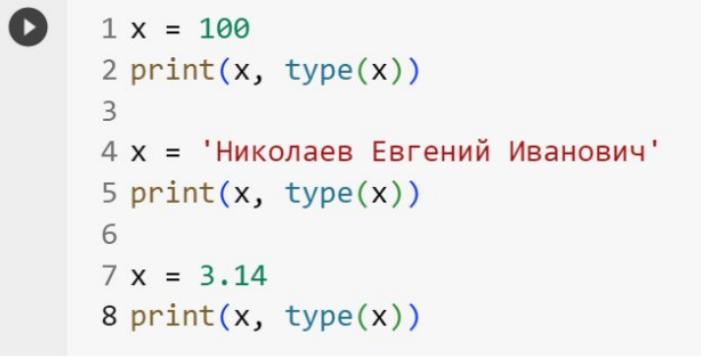
```
Evgeny <class 'str'>
Nikolaev <class 'str'>
```

2.6.6. Повторное присвоение переменной значения другого типа

Следует учитывать, что в Python переменная – это просто имя, которое ссылается на порцию данных в оперативной памяти. Этот механизм упрощает вам, программисту, хранение и получение данных. Интерпретатор Python отслеживает создаваемые вами имена переменных и порции данных, на которые эти имена переменных ссылаются. Всякий раз, когда необходимо

получить одну из этих порций данных, просто используется имя переменной, которое на эту порцию ссылается.

Переменная в Python может ссылаться на значения любого типа. После того как переменной присвоено значение одного типа, ей можно заново присвоить значение другого типа. Рассмотрим пример:



```

1 x = 100
2 print(x, type(x))
3
4 x = 'Николаев Евгений Иванович'
5 print(x, type(x))
6
7 x = 3.14
8 print(x, type(x))

```

```

100 <class 'int'>
Николаев Евгений Иванович <class 'str'>
3.14 <class 'float'>

```

2.6.7. Ввод данных

Большинство программ, выполняемых локально, должны будут читать входные данные и затем выполнять с ними операцию. Можно предоставить пользователю право вводить переменные с клавиатуры. Когда программа считывает данные с клавиатуры, она обычно сохраняет их в переменной, чтобы потом программа могла эти данные использовать.

Для чтения данных, вводимых с клавиатуры, мы будем использовать встроенную функцию Python – `input`. Функция `input` читает порцию данных, которая была введена с клавиатуры, и возвращает эту порцию данных в качестве строкового значения назад в программу. Функции `input` обычно применяют в инструкции присваивания, которая соответствует приведенному ниже общему формату:

`переменная = input(подсказка)`

В данном формате подсказка – это строковый литерал, который выводится на экран. Его предназначение – дать пользователю указание ввести

значение. А переменная – это имя переменной, которая ссылается на данные, введенные на клавиатуре. Вот пример инструкции, которая применяет функцию `input` для чтения данных с клавиатуры:

```
name = input('Как Вас зовут? ')
```

Во время исполнения этой инструкции происходит следующее:

1. Стока «Как Вас зовут?» выводится на экран.
2. Программа приостанавливает работу и ждет, когда пользователь введет что-нибудь с клавиатуры и нажмет клавишу **Enter**.
3. Когда клавиша **Enter** нажата, набранные данные возвращаются в качестве строкового значения и присваиваются переменной `name`.

В качестве демонстрации взгляните на приведенный ниже интерактивный сеанс:

Исходный код и ожидание ввода пользователя:

```
[1] 1 name = input('как вас зовут? ')
    2 print('Понятно, ', name, '. Я запомню!')
```

как вас зовут? John

Результат работы программы:

```
[10] 1 name = input('как вас зовут? ')
      2 print('Понятно, ', name, '. Я запомню!')
```

как вас зовут? John
Понятно, John . Я запомню!

2.6.8. Чтение чисел при помощи функции `input`

Функция `input` всегда возвращает введенные пользователем данные как строковые, даже если пользователь вводит числовые значения. Например, предположим, что вы вызываете функцию `input`, набираете число 72 и нажимаете клавишу <Enter>. Возвращенное из функции `input` значение будет строковым ('72', а не 72). Может возникнуть проблема, если вы захотите

использовать это значение в математической операции. Математические операции могут выполняться только с числовыми значениями, а не строковыми.

В Python имеются встроенные функции для преобразования, или конвертации, строкового типа в числовой. В табл. 2.2 приведены две такие функции.

Таблица 2.2 – Функции преобразования данных

Функция	Описание
<code>int(значение)</code>	В функцию <code>int()</code> передается аргумент, и она возвращает значение аргумента, преобразованное в целочисленный тип <code>int</code>
<code>float(значение)</code>	В функцию <code>float()</code> передается аргумент, и она возвращает значение аргумента, преобразованное в вещественный тип <code>float</code>

Предположим, что вы пишете программу расчета заработной платы и хотите получить количество часов, которое пользователь отработал. Взгляните на приведенный ниже фрагмент программного кода:

```
string_value = input('Сколько часов Вы отработали? ')
hours = int(string_value)
```

Первая инструкция получает от пользователя количество часов и присваивает его значение строковой переменной `string_value`. Вторая инструкция вызывает функцию `int()`, передавая `string_value` в качестве аргумента. Значение, на которое ссылается `string_value`, преобразуется в целочисленное `int` и присваивается переменной `hours`.

Этот пример иллюстрирует прием работы с функцией `int()`, однако он не эффективен, потому что создает две переменные: одну для хранения строкового значения, которое возвращается из функции `input()`, а другую для хранения целочисленного значения, которое возвращается из функции `int()`. Приведенный ниже фрагмент кода демонстрирует оптимальный подход. Здесь одна-единственная инструкция делает всю работу, которую ранее делали две приведенные выше инструкции, и она создает всего одну переменную:

```
hours = int(input('Сколько часов Вы проработали? '))
```

Эта инструкция использует вызов вложенной функции. Значение, которое возвращается из функции `input()`, передается в качестве аргумента в функцию `int()`. Вот как это работает:

1. Инструкция вызывает функцию `input()`, чтобы получить значение, вводимое с клавиатуры.
2. Значение, возвращаемое из функции `input()` (т. е. строковое), передается в качестве аргумента в функцию `int()`.
3. Целочисленное значение `int`, возвращаемое из функции `int()`, присваивается переменной `hours`.

После исполнения этой инструкции переменной `hours` будет присвоено введенное с клавиатуры значение, преобразованное в целочисленное `int`.

Рассмотрим еще один пример. Предположим, что вы хотите получить от пользователя почасовую ставку оплаты труда. Приведенная ниже инструкция предлагает пользователю ввести это значение с клавиатуры, преобразует это значение в вещественное `float` и присваивает его переменной `pay_rate`:

```
pay_rate = float(input('Какая почасовая ставка оплаты труда? '))
```

2.7. Выполнение расчетов

Python имеет много операторов, которые используются для выполнения математических расчетов.

Большинство реально существующих алгоритмов требует, чтобы выполнялись расчеты. Инструментами программиста для расчетов являются математические операторы. В табл. 2.3 перечислены математические операторы, которые имеются в языке Python.

Таблица 2.3 – Математические операторы

Оператор	Описание
<code>x + y</code>	Сложение
<code>x - y</code>	Вычитание
<code>x * y</code>	Умножение
<code>x / y</code>	Деление
<code>x // y</code>	Целочисленное деление

$x^{**} y$	Возведение в степень (x в степень y)
$x \% y$	Остаток от деления x на y
$-x$	Унарный минус
$+x$	Унарный плюс

Значения справа и слева от операторов называются operandами. Операторы позволяют производить вычисления как с литерами, так и с переменными.

Пример:

```
1 # Зарплата
2 salary = 15000.00
3 # Премия
4 bonus = 13500.80
5 # Ставка налога
6 tax = 13
7 # Сумма налога
8 total_tax = (salary + bonus) * tax / 100
9 # итоговая выплата
10 total_outcome = (salary + bonus) - total_tax
11 print('Выплата на руки', total_outcome)
```

Выплата на руки 24795.696

В представленном листинге в строках 8 и 10 производится вычисление итоговой выплаты с использованием арифметических операторов. Operandами выступают переменные **salary**, **bonus**, **tax** и числовой литерал 100. В представленном примере возникла не совсем корректная ситуация: несмотря на то, что расчет произведен верно, итоговый результат не совсем корректен с точки зрения предметной области. Результат, на который ссылается переменная **total_outcome** равен 24795.696 (24795 рублей 696 копеек). Но в реальной жизни денежные суммы указываются с точностью до сотых (до копеек), то есть необходимо было бы получить результат 24795.70 (по правилам округления).

В Python данные манипуляции можно реализовать различными способами. Мы воспользуемся встроенными функциями Python (таблица 2.4).

Таблица 2.4 – Встроенные математические функции

Оператор	Описание
----------	----------

<code>abs(x)</code>	Возвращает модуль x
<code>divmod(x, y)</code>	Возвращает $(x//y, x\%y)$
<code>pow(x,y)</code>	Возвращает $x^{**}y$
<code>round(x,[n])</code>	Округляет до ближайшего кратного 10 в степени n (банковское округление)

Предыдущий листинг можно переписать следующим образом:

```
▶ 1 salary = 15000.00
  2 bonus = 13500.80
  3 tax = 13
  4 total_tax = (salary + bonus) * tax / 100
  5 total_outcome = (salary + bonus) - total_tax
  6 total_outcome = round(total_outcome, 2)
  7 print('Выплата на руки', total_outcome)
```

⇨ Выплата на руки 24795.7

В строке 6 применяется функция `round()`. Теперь результирующая выплата равна 24795 рублей 70 копеек.

Вопросы для самопроверки по теме 2

1. Кто является клиентом программиста?
2. Что такое техническое требование к программному обеспечению?
3. Что такое алгоритм?
4. Что такое псевдокод?
5. Что такое блок-схема?
6. Что означают приведенные ниже символы блок-схемы?
 - Овал.
 - Параллелограмм.
 - Прямоугольник
7. Напишите инструкцию, которая показывает ваше имя.
8. Напишите инструкцию, которая показывает приведенный ниже текст: **Python – лучше всех!**
9. Напишите инструкцию, которая показывает приведенный ниже текст: **Кошка сказала "мяу"**.

10. Что такое переменная?

11. Какие из приведенных ниже имен переменных недопустимы в Python и почему?

99bottles
july2009
theSalesFigureForFiscalYear
r&d
grade_report

12. Являются ли имена переменных **Sales** и **sales** одинаковыми?
 Почему?

13. Допустима ли приведенная ниже инструкция присваивания? Если она недопустима, то почему?

72 = amount

14. Что покажет приведенный ниже фрагмент кода?

```
val = 99
print('Значение равняется', 'val')
```

15. Взгляните на приведенные ниже инструкции присваивания:

```
value1 = 99
value2 = 45.9
value3 = 7.0
value4 = 7
value5 = 'abc'
```

Какой тип данных Python будут иметь эти значения, когда на них будут ссылаться переменные после исполнения указанных инструкций?

16. Что покажет приведенный ниже фрагмент кода?

my_value = 99

```
my_value = 0  
print(my_value)
```

17. Вам нужно, чтобы пользователь программы ввел фамилию клиента. Напишите инструкцию, которая предлагает пользователю ввести эти данные и присваивает их переменной.

18. Вам нужно, чтобы пользователь программы ввел объем продаж за неделю. Напишите инструкцию, которая предлагает пользователю ввести эти данные и присваивает их переменной.

Лекция 3. Управление ходом выполнения программы

План лекции

1. Условный оператор.
2. Структуры повторения.
3. Вычисления с использованием циклов.

3.1. Условный оператор

Инструкция `if` применяется для создания управляющей структуры, которая позволяет иметь в программе более одного пути исполнения. Инструкция `if` исполняет одну или несколько инструкций, только когда булево выражение является истинным.

Управляющая структура – это логическая схема, управляющая порядком, в котором исполняется набор инструкций. До сих пор мы использовали только самый простой тип управляющей структуры: последовательную структуру, т. е. структуру с последовательным исполнением. Последовательная структура представляет собой набор инструкций, которые исполняются в том порядке, в котором они появляются. Например, приведенный ниже фрагмент кода имеет последовательную структуру, потому что инструкции исполняются сверху вниз:

```
 1 # Количество отработанных часов
 2 hours = 25
 3 # Стоимость часа работы
 4 tax = 550.37
 5
 6 # Итоговый заработка
 7 outcome = hours * tax
 8
 9 print('Итого к выплате: ', outcome)
```

Итого к выплате: 13759.25

Но что если последовательной структуры программы недостаточно? Например, в зависимости от количества отработанных часов, работнику будет начислена премия... Преобразуем предыдущую задачу к виду: если работник

отработал не менее 40 часов, то начисляется премия в 10 процентов от суммарного заработка. Код изменится следующим образом:

```
1 # Количество отработанных часов
2 hours = 40
3 # Стоимость часа работы
4 tax = 550.37
5
6 # Итоговый заработка
7 outcome = hours * tax
8 if hours >= 40:
9     outcome = outcome + 0.1 * outcome
10
11 outcome = round(outcome, 2)
12 print('Итого к выплате: ', outcome)
```

Итого к выплате: 24216.28

В строке 7 вычисляется выплата без учета премии, но в строке 8 ставится условие **hours >= 40**, и, если это условие выполнено, то значение переменной **outcome** увеличивается на 10 процентов. На рис. 3.1 показана блок-схема для оператора **if**.



Рисунок 3.1 – Простая структура принятия решения

Программисты называют показанный на рис. 3.1 тип структуры структурой принятия решения с единственным вариантом. Это связано с тем, что она предоставляет всего один вариант пути исполнения. Если условие в ромбовидном символе истинное, то мы принимаем этот вариант пути. В противном случае мы выходим из этой структуры. На рис. 3.2 представлен более детализированный пример, в котором выполняются три действия, только когда количество отработанных часов не менее 40.

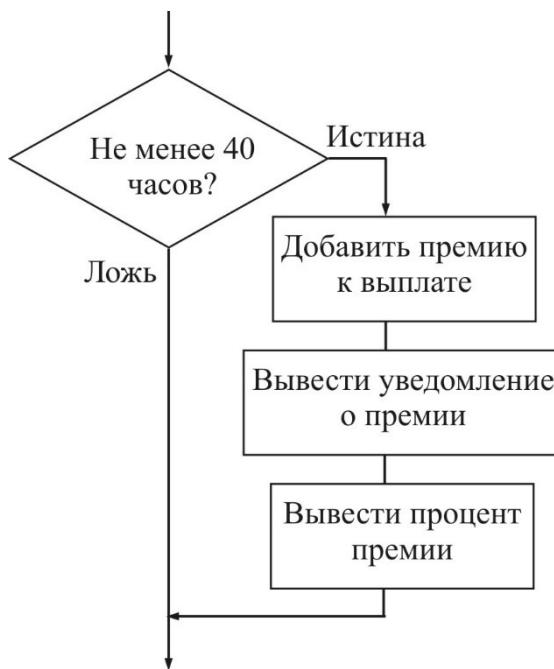


Рисунок 3.2 – Структура принятия решения, которая исполняет три действия при условии, что количество отработанных часов не менее 40

Исходный код для представленной блок-схемы следующий:

```

1 # Количество отработанных часов
2 hours = 40
3 # Стоимость часа работы
4 tax = 550.37
5
6 # Итоговый заработка
7 outcome = hours * tax
8 if hours >= 40:
9     outcome = outcome + 0.1 * outcome
10    print('Премия начислена!!!')
11    print('Премия в размере 10%')
12
13 outcome = round(outcome, 2)
14 print('Итого к выплате: ', outcome)
  
```

Премия начислена!!!

Премия в размере 10%

Итого к выплате: 24216.28

Это по-прежнему структура принятия решения с единственным вариантом, потому что имеется всего один вариант пути исполнения. В Python для написания структуры принятия решения с единственным вариантом используется инструкция **if**. Вот общий формат инструкции **if**:

if условие:

инструкция

инструкция

Для простоты мы будем называть первую строку условным выражением, или выражением ***if***. Условное выражение начинается со слова ***if***, за которым следует условие, т. е. выражение, которое будет вычислено, как истина либо ложь. После условия стоит двоеточие. Со следующей строки начинается блок инструкций. Блок – это просто набор инструкций, которые составляют одну группу. Обратите внимание, что в приведенном выше общем формате все инструкции блока выделены отступом. Такое оформление кода обязательно, потому что интерпретатор Python использует отступы для определения начала и конца блока.

Во время исполнения инструкции ***if*** осуществляется проверка условия. Если условие истинное, то исполняются инструкции, которые появляются в блоке после условного выражения. Если условие ложное, то инструкции в этом блоке пропускаются.

3.1.1. Булевые выражения и операторы сравнения

Как упоминалось ранее, инструкция ***if*** осуществляет проверку выражения, чтобы определить, является ли оно истинным или ложным. Выражения, которые проверяются инструкцией ***if***, называются булевыми выражениями в честь английского математика Джорджа Буля.

В 1800-х годах Буль изобрел математическую систему, в которой абстрактные понятия истинности и ложности могли использоваться в вычислениях.

Как правило, булево выражение, которое проверяется инструкцией ***if***, формируется оператором сравнения (реляционным оператором). Оператор сравнения определяет, существует ли между двумя значениями определенное отношение. Например, оператор больше (***>***) определяет, является ли одно значение больше другого. Оператор равно (***==***) – равны ли два значения друг другу. В табл. 3.1 перечислены имеющиеся в Python операторы сравнения.

Таблица 3.1 – Операторы сравнения

Оператор	Значение
>	Больше
<	Меньше
\geq	Больше или равно
\leq	Меньше или равно
$=$	Равно
\neq	Не равно

3.1.2. Инструкция if-else

Инструкция **if-else** исполняет один блок инструкций, если ее условие является истинным, либо другой блок, если её условие является ложным.

Рассмотрим структуру принятия решения с двумя альтернативными вариантами, в которой имеется два возможных пути исполнения – один путь принимается, если условие является истинным, и другой путь принимается, если условие является ложным. На рис. 3.3 представлена блок-схема для структуры принятия решения с двумя альтернативными вариантами

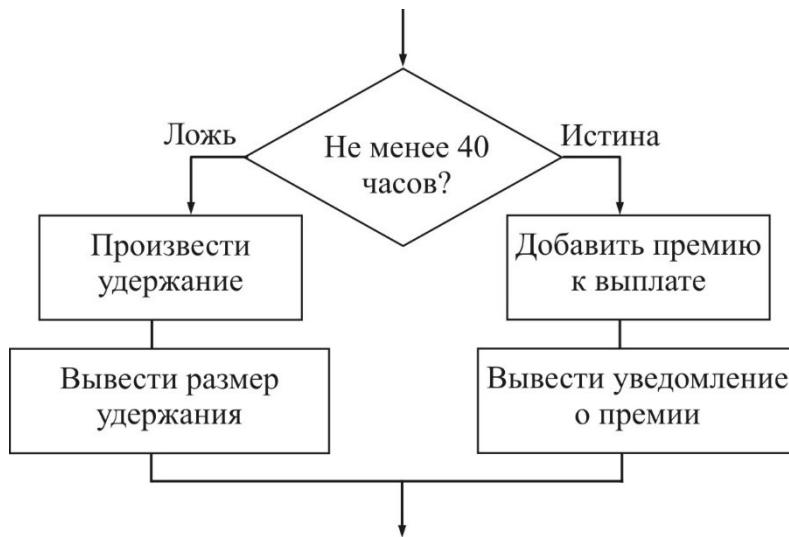


Рисунок 3.3 – Структура принятия решения с двумя альтернативными вариантами

Исходный код, соответствующий данной блок-схеме:

```

1 # Количество отработанных часов
2 hours = 38
3 # Стоимость часа работы
4 tax = 550.37
5
6 outcome = hours * tax
7 if hours >= 40:
8     outcome = outcome + 0.2 * outcome
9     print('Премия начислена!!! Размер премии 20%')
10 else:
11     outcome = outcome - 1000
12     print('Удержано 1000 рублей')
13
14 outcome = round(outcome, 2)
15 print('Итого к выплате: ', outcome)

```

Удержано 1000 рублей
Итого к выплате: 19914.06

В программном коде мы записываем структуру принятия решения с двумя альтернативными вариантами, как инструкцию **if-else**. Вот общий формат инструкции **if-else**:

if условие:

инструкция

инструкция

else:

инструкция

инструкция

Когда эта инструкция исполняется, осуществляется проверка условия. Если оно истинное, то исполняется блок инструкций с отступом, расположенный после условного выражения, затем поток управления программы перескакивает к инструкции, которая следует за инструкцией **if-else**. Если условие ложное, то исполняется блок инструкций с отступом, расположенный после выражения **else**, затем поток управления программы перескакивает к инструкции, которая следует за инструкцией **if-else**.

Когда вы пишете инструкцию **if-else**, при выделении отступами следует руководствоваться следующими принципами:

- ◆ убедитесь, что выражение **if** и выражение **else** выровнены относительно друг друга;

- ♦ выражение `if` и выражение `else` сопровождаются блоком инструкций. Убедитесь, что инструкции в блоках расположены с одинаковым отступом.

3.1.3. Вложенные структуры принятия решения и инструкция if-elif-else

Для проверки более одного условия структура принятия решения может быть вложена внутрь другой структуры принятия решения. Рассмотрим использование `if-elif-else` на примере задачи: сотруднику начисляется сдельная заработка плата (количество отработанных часов умножается на стоимость одного часа). Но затем начисляется премия по следующим правилам:

Отработано времени, часы	Размер премии (штрафа), %
0-9	-10
10-19	0
20-29	10
30-39	20
>40	30

Для решения данной задачи разработан исходный код на основе применения конструкции `if-else`:

```
1 # Количество отработанных часов
2 hours = 32
3 # Стоимость часа работы
4 tax = 400
5
6 outcome = hours * tax
7 if hours >= 0 and hours < 9:
8     outcome = outcome - 0.2 * outcome
9     print('Начислен штраф в размере 10%')
10 else:
11     if hours >= 10 and hours < 19:
12         print('Нет ни премий, ни удержаний')
13     else:
14         if hours >= 20 and hours < 29:
15             outcome = outcome + 0.1 * outcome
16             print('Начислена премия в размере 10%)')
```

```

17     else:
18         if hours >= 30 and hours < 39:
19             outcome = outcome + 0.2 * outcome
20             print('Начислена премия в размере 20%')
21         else:
22             if hours >= 40:
23                 outcome = outcome + 0.3 * outcome
24                 print('Начислена премия в размере 30%')
25 outcome = round(outcome, 2)
26 print('Итого к выплате: ', outcome)

```

Начислена премия в размере 20%
Итого к выплате: 15360.0

Код получился достаточно громоздкий, он трудно воспринимается, поэтому легко допустить ошибку. Для переработки исходного кода в более простой для понимания, программирования необходимо воспользоваться конструкцией **if-elif-else**:

```

1 # Количество отработанных часов
2 hours = 32
3 # Стоимость часа работы
4 tax = 400
5 outcome = hours * tax
6 if hours >= 0 and hours < 9:
7     outcome = outcome - 0.2 * outcome
8     print('Начислен штраф в размере 10%')
9 elif hours < 19:
10    print('Нет ни премий, ни удержаний')
11 elif hours < 29:
12    outcome = outcome + 0.1 * outcome
13    print('Начислена премия в размере 10%')
14 elif hours < 39:
15    outcome = outcome + 0.2 * outcome
16    print('Начислена премия в размере 20%')
17 else:
18    outcome = outcome + 0.3 * outcome
19    print('Начислена премия в размере 30%')
20 outcome = round(outcome, 2)
21 print('Итого к выплате: ', outcome)

```

▶ Начислена премия в размере 20%
Итого к выплате: 15360.0

Python предоставляет специальный вариант структуры принятия решения, именуемый инструкцией **if-elif-else**, которая упрощает написание

```

логической конструкции такого типа. Вот общий формат инструкции if-elif-
else:

    if условие_1:
        инструкция
        инструкция
        ...
    elif условие_2:
        инструкция
        инструкция
        ...
# Вставить столько выражений elif, сколько нужно:
else:
    инструкция
    инструкция
    ...

```

При исполнении этой инструкции проверяется **условие_1**. Если оно является истинным, то исполняется блок инструкций, который следует сразу после него, вплоть до выражения **elif**.

Остальная часть структуры игнорируется. Однако если **условие_1** является ложным, то программа перескакивает непосредственно к следующему выражению **elif** и проверяет **условие_2**. Если оно истинное, то исполняется блок инструкций, который следует сразу после него, вплоть до следующего выражения **elif**. И остальная часть структуры тогда игнорируется. Этот процесс продолжается до тех пор, пока не будет найдено условие, которое является истинным, либо пока больше не останется выражений **elif**. Если ни одно условие не является истинным, то исполняется блок инструкций после выражения **else**.

При программировании логических выражений в условных конструкциях часто используются логические операторы **or**, **and** и **not** (таблица 3.2)

Таблица 3.2 – Логические операторы

Оператор	Значение
and	Оператор соединяет два булева выражения в одно составное выражение. Для того чтобы составное выражение было истинным, оба подвыражения должны быть истинными
or	Оператор соединяет два булева выражения в одно составное выражение. Для того чтобы составное выражение было истинным, одно либо оба подвыражения должны быть истинными. Достаточно, чтобы только одно из выражений было истинным, и не имеет значения какое из них
not	Оператор является унарным оператором, т. е. он работает только с одним операндом. Операнд должен быть булевым выражением. Оператор not инвертирует истинность своего операнда. Если он применен к выражению, которое является истинным, то этот оператор возвращает ложь. Если он применен к выражению, которое является ложным, то этот оператор возвращает истину

3.2. Структуры повторения

Структура с повторением исполняет инструкции или набор инструкций многократно. Программистам в большинстве случаев приходится писать код, который выполняет одну и ту же задачу снова и снова. Предположим, необходимо написать программу, которая вычисляет объем воды не в одном, а в нескольких баках различного размера. Один из подходов, который предоставляет неплохое решение, состоит в том, чтобы написать программный код, вычисляющий объем для одного бака, а затем повторить этот программный код для каждого бака. Пример кода:

```

1 h = float(input('высота бака, см > '))
2 r = float(input('радиус основания бака, см > '))
3 v = 3.14 * (r**2) * h
4 print('Объем бака', v, 'куб. см')
5 print('*' * 50)
6
7 h = float(input('высота бака, см > '))
8 r = float(input('радиус основания бака, см > '))
9 v = 3.14 * (r**2) * h
10 print('Объем бака', v, 'куб. см')
11 print('*' * 50)
12

```

```

13 h = float(input('высота бака, см > '))
14 r = float(input('радиус основания бака, см > '))
15 v = 3.14 * (r**2) * h
16 print('Объем бака', v, 'куб. см')
17 print('*' * 50)

```

```

→ высота бака, см > 1
радиус основания бака, см > 2
Объем бака 12.56 куб. см
*****
высота бака, см > 3
радиус основания бака, см > 4
Объем бака 150.72 куб. см
*****
высота бака, см > 4
радиус основания бака, см > 9
Объем бака 1017.36 куб. см
*****

```

И этот код повторяется снова и снова... Как видите, этот программный код представляет собой одну длинную последовательную структуру, содержащую много повторяющихся фрагментов. У этого подхода имеется несколько недостатков:

- ◆ повторяющийся код увеличивает программу;
- ◆ написание длинной последовательности инструкций может быть трудоемким;
- ◆ если часть повторяющегося программного кода нужно исправить или изменить, то исправление или изменение должны быть повторены много раз.

Вместо неоднократного написания одинаковой последовательности инструкций более оптимальный способ неоднократно выполнить операцию состоит в том, чтобы написать программный код операции всего один раз и затем поместить этот код в структуру, которая предписывает компьютеру повторять его столько раз, сколько нужно. Это делается при помощи структуры с повторением, которая более широко известна как цикл.

3.2.1. Цикл с условием повторения

Цикл с условием повторений использует логическое условие со значениями истина/ложь, которое управляет количеством повторов цикла. Для

написания цикла с условием повторения в Python применяется инструкция **while**.

Цикл **while** («пока») получил свое название из-за характера своей работы: он выполняет некую задачу до тех пор, пока условие является истинным. Данный цикл имеет две части: **условие**, которое проверяется на истинность либо ложность, и инструкцию или набор инструкций, которые повторяются до тех пор, пока условие является истинным. На рис. 3.4 представлена логическая схема цикла **while**.

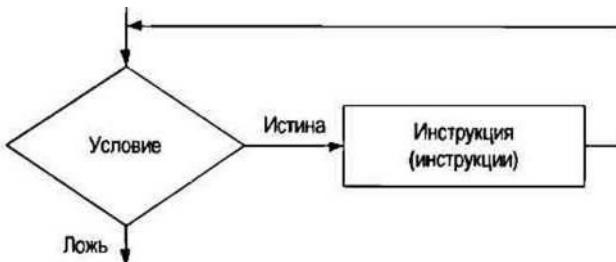


Рисунок 3.4 – Логическая схема цикла **while**

Ромбовидный символ представляет проверяемое условие. Обратите внимание, что происходит, если условие является истинным: исполняются одна или несколько инструкций, и выполнение программы перетекает назад к точке чуть выше ромба. Условие проверяется снова, и если оно истинное, то процесс повторяется. Если условие является ложным, программа выходит из цикла. В блок-схеме цикл всегда идентифицируется по соединительной линии, которая возвращается в предыдущую часть блок-схемы.

Вот общий формат цикла с условием повторения в Python:

while условие:

инструкция

инструкция

Для простоты мы будем называть первую строку выражением **while**. Оно начинается со слова **while**, после которого идет булево условие, вычисляемое как истина либо как ложь. После условия идет двоеточие. Начиная со следующей строки, расположена блок инструкций. (понятно, что все инструкции в блоке должны быть единообразно выделены отступом. Такое

выделение отступом необходимо потому, что интерпретатор Python использует его для определения начала и конца блока).

При исполнении цикла `while` проверяется условие. Если условие является истинным, то исполняются инструкции, которые расположены в блоке после выражения `while`, и цикл начинается сначала. Если условие является ложным, то программа выходит из цикла.

Теперь преобразуем программу вычисления объема баков, используя новые знания о цикле с условием:

```
▶ 1 keep_going = 'Y'
  2 while keep_going == 'Y' or keep_going == 'y':
  3     h = float(input('высота бака, см > '))
  4     r = float(input('радиус основания бака, см > '))
  5     v = 3.14 * (r**2) * h
  6     print('Объем бака', v, 'куб. см')
  7     print('*' * 50)
  8     keep_going = input('Вычислим еще один? (Y или у - продолжим) > ')
```

⇨ высота бака, см > 1
 радиус основания бака, см > 2
 Объем бака 12.56 куб. см

 Вычислим еще один? (Y или у - продолжим) > у
 высота бака, см > 5
 радиус основания бака, см > 6
 Объем бака 565.2 куб. см

 Вычислим еще один? (Y или у - продолжим) > Y
 высота бака, см > 5
 радиус основания бака, см > 5
 Объем бака 392.5 куб. см

 Вычислим еще один? (Y или у - продолжим) > у
 высота бака, см > 7
 радиус основания бака, см > 7
 Объем бака 1077.02 куб. см

 Вычислим еще один? (Y или у - продолжим) > n

Цикл `while` также называется циклом с предусловием, а именно он проверяет свое условие до того, как сделает итерацию. Поскольку проверка осуществляется в начале цикла, обычно нужно выполнить несколько шагов перед началом цикла, чтобы гарантировать, что цикл выполнится как минимум однажды.

Всегда, кроме редких случаев, циклы должны содержать возможность завершиться. То есть в цикле что-то должно сделать проверяемое условие ложным. Цикл в представленной выше программе код завершается, когда

выражение `keep_going` перестает быть равным '`y`' или '`Y`' ным. Если цикл не имеет возможности завершиться, он называется бесконечным циклом. Бесконечный цикл продолжает повторяться до тех пор, пока программа не будет прервана. Бесконечные циклы обычно появляются, когда программист забывает написать программный код внутри цикла, который делает проверяемое условие ложным. В большинстве случаев следует избегать применения бесконечных циклов.

3.2.2. Цикл `for`: цикл со счетчиком повторений

Цикл со счетчиком повторений повторяется заданное количество раз. В Python для написания цикла со счетчиком повторений применяется инструкция `for`.

Для написания цикла со счетчиком повторений применяется инструкция `for`. В Python инструкция `for` предназначена для работы с последовательностью значений данных. Когда эта инструкция исполняется, она повторно выполняется для каждого значения последовательности. Вот ее общий формат:

```
for переменная in [значение1, значение2, ...]:
    инструкция
    инструкция
```

Мы будем обозначать первую строку, как выражение `for`. В выражении `for` переменная – это имя переменной. Внутри скобок находится последовательность разделенных запятыми значений (в Python последовательность разделенных запятыми значений данных, заключенная в скобки [], называется списком).

Инструкция `for` исполняется следующим образом: переменной присваивается первое значение в списке, и затем исполняются инструкции, которые расположены в блоке. Далее переменной присваивается следующее значение в списке, и инструкции в блоке исполняются снова. Этот процесс

продолжается до тех пор, пока переменной не будет присвоено последнее значение в списке.

Рассмотрим пример программы, которая выполняет блок операторов ровно 6 раз – по количеству элементов в списке цикла **for**:

```
 1 for x in [1,3,4,5,6,78]:  
2   print('*' * 20)  
3   print(x, '\t', x*x)
```

```
*****  
1      1  
*****  
3      9  
*****  
4      16  
*****  
5      25  
*****  
6      36  
*****  
78     6084
```

Во время первой итерации цикла **for** переменной **x** присваивается значение 1, и затем исполняется инструкция в строке 2 (печатает значение 1 и квадрат 1^2). Во время следующей итерации цикла переменной **x** присваивается значение 3, и исполняется инструкция в строке 2 (печатается 3 и квадрат 3^2). Как показано в листинге, этот процесс продолжается до тех пор, пока переменной **x** не присваивается последнее значение в списке. Поскольку в списке всего шесть значений, цикл сделает шесть итераций.

Программисты Python обычно называют переменную, которая используется в выражении **for**, целевой переменной, потому что она является целью присвоения в начале каждой итерации цикла.

Значения в списке **for** могут быть любого типа, например:

```

1 for x in [2,'abc', True, 4.55, 5, 'Hello']:
2   print('*' * 20)
3   print(x, '\t', x*x)

*****
2      4
*****
```

```

TypeError                                 Traceback (most recent call last)
<ipython-input-25-d6896f3424e9> in <cell line: 1>()
      1 for x in [2,'abc', True, 4.55, 5, 'Hello']:
      2   print('*' * 20)
----> 3   print(x, '\t', x*x)

TypeError: can't multiply sequence by non-int of type 'str'
```

В данном скрипте произошла ошибка, так как в теле цикла производится вычисление квадрата каждого элемента списка, но вычислить квадрат для строковых и логических значений невозможно. Исправим программу следующим образом:

```

1 for x in [2,'abc', True, 4.55, 5, 'Hello']:
2   print('*' * 20)
3   if type(x)==int or type(x)==float:
4     print(x, '\t', x*x)
5   else:
6     print(x)

*****
2      4
*****
abc
*****
True
*****
4.55      20.70249999999997
*****
5      25
*****
Hello
```

В данном листинге в строке 3 в теле цикла производится проверка каждого значения из списка: если значение целочисленного или вещественного типа, то производится вывод значения x и квадрата значения $x*x$, если нет – только значения x .

Python предоставляет встроенную функцию **range** (диапазон), которая упрощает процесс написания цикла со счетчиком повторений. Функция **range** создает тип объекта, который называется итерируемым, т. е. пригодным для итеративной обработки в цикле. Итерируемый объект аналогичен списку. Он содержит последовательность значений, которые можно по порядку обойти на

основе чего-то наподобие цикла. Вот пример для цикла, который применяет функцию `range`:

```
 1 for e in range(20):  
 2   print(e, end = ', ')  
  
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
```

В этой инструкции функция `range` порождает итерируемую последовательность целых чисел в диапазоне от 0 (включительно) до 20 (не включительно).

Функцию `range` можно использовать для задания управляющей последовательности с переопределенным начальным значением:

```
 1 for e in range(14, 20):  
 2   print(e, end = ', ')  
  
14, 15, 16, 17, 18, 19,
```

В функцию `range` можно передать еще третий аргумент, который будет рассматриваться как величина шага:

```
 1 for e in range(10, 100, 20):  
 2   print(e, end = ', ')  
  
10, 30, 50, 70, 90,
```

Таким образом, в инструкции `for` в функцию `range` передаются три аргумента:

- ◆ первый аргумент – начальное значение последовательности;
- ◆ второй аргумент – конечный предел списка;
- ◆ третий аргумент – величина шага.

3.3. Вычисления с использованием циклов

Рассмотрим практические задачи, которые реализуются с использованием циклов.

Задача 1. Напишите программу, которая для каждого месяца года выводит объем продаж нарастающим итогом с начала года. Объем продаж в зависимости от номера месяца прогнозируется по формуле:

Объем_продаж = Номер_месяца * 1000 / (Номер_месяца + 1)

Решение данной задачи реализуем с использованием следующего кода:

```
▶ 1 total = 0
 2 print('Месяц \t Продажи \t Накопительный итог')
 3 print('-'*50)
 4 for i in range(12):
 5     month_number = i+1 # Вычисляем номер месяца
 6     money = month_number * 1000 / (month_number + 1)
 7     total = total + money
 8     print(month_number, '\t', round(money, 2), '\t', round(total, 2))
```

→	Месяц	Продажи	Накопительный итог
	1	500.0	500.0
	2	666.67	1166.67
	3	750.0	1916.67
	4	800.0	2716.67
	5	833.33	3550.0
	6	857.14	4407.14
	7	875.0	5282.14
	8	888.89	6171.03
	9	900.0	7071.03
	10	909.09	7980.12
	11	916.67	8896.79
	12	923.08	9819.87

В строке 1, до цикла, происходит инициализация переменной **total**, которая выступает в качестве накопителя. Это связано с тем, что в строке 7 происходит добавление очередного месячного значения продаж к уже полученному объему поступлений. Если переменная **total** к этому времени не будет инициализирована, то произойдет ошибка.

Задача 2. Напишите функцию вычисления факториала числа.

Факториал числа **N** обозначается как **N!** и вычисляется по формуле:

$$N! = 1 * 2 * 3 * 4 * 5 * \dots * N$$

Для решения данной задачи используем следующий код:

```
▶ 1 N = int(input('Введите целое число N > '))
 2 N_fact = 1
 3 while N>0:
 4     N_fact = N_fact * N
 5     N = N - 1
 6
 7 print('N! = ', N_fact)
```

→ Введите целое число N > 25
 N! = 15511210043330985984000000

Следует обратить внимание на значения, которые присваиваются переменным-накопителям в задачах 1 и 2.

Вопросы для самопроверки по теме 3

1. Что такое управляющая структура?
2. Что такое структура принятия решения?
3. Что такое структура принятия решения с единственным вариантом исполнения?
4. Что такое булево выражение?
5. Какие типы отношений между значениями можно проверить при помощи операторов сравнения (реляционных операторов)?
6. Напишите инструкцию **if**, которая присваивает значение 0 переменной **x**, если переменная **y** равна 20.
7. Напишите, инструкцию **if**, которая присваивает значение 0.2 переменной **commissionRate** (ставка комиссионного вознаграждения), если переменная **sales** (продажи) больше или равна 10 000.
8. Как работает структура принятия решения с двумя альтернативными вариантами?
9. Какую инструкцию следует применить в Python для написания структуры принятия решения с двумя альтернативными вариантами?
10. При каких обстоятельствах срабатывают инструкции, которые появляются после выражения **else** при написании инструкции **if-else**?
11. Преобразуйте приведенный ниже фрагмент кода в инструкцию **if-elif-else**:

```

if number == 1:
    print('Один')
else:
    if number == 2:
        print('Два')
    else:

```

```

if number == 3:
    print('Три')
else:
    print('Неизвестное')

```

12. Объясните, что имеется в виду под термином «исполняемый по условию».

13. Вам нужно проверить условие. Если оно является истинным, то выполнить один набор инструкций. Если же оно является ложным, то выполнить другой набор инструкций. Какую структуру вы будете использовать?

14. Кратко опишите, как работает оператор `and`.
15. Кратко опишите, как работает оператор `or`.
16. Какой логический оператор лучше всего использовать при определении, находится ли число внутри диапазона?
17. Что такое флаг и как он работает?
18. Что такое итерация цикла?
19. Когда цикл `while` проверяет свое условие: до или после того, как он выполнит итерацию?
20. Сколько раз сообщение '`Привет, мир!`' будет напечатано в приведенном ниже фрагменте кода?

```

count = 10
while count < 1:
    print('Привет, мир!')

```

21. Что такое бесконечный цикл?
 22. Перепишите приведенный ниже фрагмент кода, чтобы вместо использования списка `[0, 1, 2, 3, 4, 5]` он вызывал функцию `range`:
- ```

for x in [0, 1, 2, 3, 4, 5]:
 print('Обожаю эту программу!')

```

23. Что покажет приведенный ниже фрагмент кода?

```
for number in range(6):
 print(number)
```

24. Что покажет приведенный ниже фрагмент кода?

```
for number in range(2, 6):
 print(number)
```

25. Что покажет приведенный ниже фрагмент кода?

```
for number in range(0, 501, 100):
 print(number)
```

26. Что покажет приведенный ниже фрагмент кода?

```
for number in range(10, 5, -1):
 print(number)
```

27. Что такое накопитель (аккумуляторная переменная)?

28. Следует ли инициализировать накопитель конкретным значением? Почему или почему нет?

29. Что покажет приведенный ниже фрагмент кода?

```
total = 0
for count in range(1, 6):
 total = total + count
print(total)
```

30. Что покажет приведенный ниже фрагмент кода?

```
Number1 = 10
number2 = 5
number1 = number1 + number2
print(number1, number2)
```

31. Перепишите приведенные ниже инструкции с использованием расширенных операторов присваивания:

```
quantity = quantity + 1
```

```
days_left = days_left - 5
price = price * 10
price = price / 2
```

32. Что такое цикл с условием повторения?
33. Что такое цикл со счетчиком повторений?
34. Что такое бесконечный цикл? Напишите программный код для бесконечного цикла.
35. Почему соответствующая инициализация накапливающих переменных имеет критически важное значение?

## Лекция 4. Функции

### План лекции

1. Процедуры и функции.
2. Объявление и использование функций.
3. Область действия и локальные переменные.
4. Передача аргументов в функцию.
5. Возврат значений из функции.
6. Функции стандартной библиотеки и инструкция import
7. Рекурсия.
8. Функции в вычислительных задачах.

### 4.1. Процедуры и функции

Функция – это группа инструкций (со своим именем), которая существует внутри программы с целью выполнения определенной задачи.

Большинство программ выполняет задачи, которые настолько крупные, что их приходится разбивать на несколько подзадач. Поэтому программисты обычно подразделяют свои программы на небольшие приемлемые порции, которые называются функциями. Функция – это группа инструкций, которая существует внутри программы с целью выполнения конкретной задачи. Вместо того чтобы писать большую программу как одну длинную последовательность инструкций, программист создает несколько небольших функций, каждая из которых выполняет определенную часть задачи. Эти небольшие функции затем могут быть исполнены в нужном порядке для выполнения общей задачи.

Такой подход иногда называется методом "разделяй и властвуй", потому что большая задача подразделяется на несколько меньших задач, которые легко выполнить. На рис. 4.1 иллюстрируется эта идея путем сравнения двух программ: в одной из них для выполнения задачи используется длинная сложная последовательность инструкций, в другой задача подразделяется на

меньшие по объему задачи, каждая из которых выполняется отдельной функцией.



Рисунок 4.1 – Использование функций для разбиения задачи по методу «разделяй и властвуй»

При использовании в программе функций каждая подзадача обычно выносится в собственную функцию. Следует понимать, что простое разбиение последовательной программы на блоки не является самоцелью. Функции предполагают повторное использование, то есть созданные единожды функции могут вызываться любое количество раз. Например, однажды созданная разработчиками функция `print` используется программистами многоократно, каждый раз, когда необходимо что-либо вывести. То же относится к встроенным функциям `abs( )`, `round( )`, `range( )` и т.д.

В результате разбиения программы на функции она получает следующие преимущества.

- ◆ Более простой код. Когда код программы разбит на функции, он проще и легче для понимания. Несколько небольших функций намного легче читать, чем одну длинную последовательность инструкций.
- ◆ Повторное использование кода. Функции также уменьшают дублирование программного кода в программе. Если определенная операция в программе выполняется в не скольких местах, то для выполнения этой операции можно один раз написать функцию и затем ее выполнять в любое время, когда она понадобится. Это преимущество функций называется повторным использованием кода.
- ◆ Более простое тестирование. Когда каждая задача в программе содержится в собственной функции, процессы тестирования и отладки становятся проще. Программисты могут индивидуально протестировать каждую функцию в программе и определить, выполняет ли она свою задачу правильно. Это упрощает процесс изолирования и исправления ошибок.
- ◆ Более быстрая разработка. Предположим, что программист или команда программистов разрабатывают многочисленные программы. Они обнаруживают, что каждая программа выполняет несколько общих задач, таких как выяснение имени пользователя и пароля, вывод текущего времени и т. д. Многократно писать программный код для всех этих задач не имеет никакого смысла. Вместо этого для часто встречающихся задач пишут функции, и эти функции могут быть включены в состав любой программы, которая в них нуждается.
- ◆ Упрощение командной работы. Функции также упрощают программистам работу в командах. Когда программа разрабатывается как набор функций, каждая из которых выполняет отдельную задачу, в этом случае разным программистам может быть поручено написание различных функций.

Существуют несколько типов функций: функции без возврата значения, или процедуры (procedure), и функции с возвратом значения (function). Когда вызывается функция без возврата значения, она просто исполняет

содержащиеся в ней инструкции и затем завершается. Когда вызывается функция с возвратом значения, она исполняет содержащиеся в ней инструкции и возвращает значение в ту инструкцию, которая ее вызвала. Функция `input()` является примером функции с возвратом значения. При вызове функции `input()` она получает данные, которые пользователь вводит на клавиатуре, и возвращает эти данные в качестве строкового значения. Функции `int()` и `float()` – тоже примеры функций с возвратом значения. Вы передаете аргумент функции `int()`, и она возвращает значение этого аргумента, преобразованное в целое число. Аналогичным образом передается аргумент функции `float()`, и она возвращает значение этого аргумента, преобразованное в число с плавающей точкой.

## **4.2. Объявление и использование функций**

Программный код функции называется определением функции. Для исполнения функции пишется инструкция, которая ее вызывает.

Имена функциям назначаются точно так же, как назначаются имена используемым в программе переменным. Имя функции должно быть достаточно описательным, чтобы любой читающий ваш код мог обоснованно догадаться, что именно функция делает.

В Python требуется, чтобы соблюдались такие же правила, которые вы соблюдаете при именовании переменных:

- ◆ в качестве имени функции нельзя использовать одно из ключевых слов Python;
- ◆ имя функции не может содержать пробелы;
- ◆ первый символ должен быть одной из букв от a до z, от A до Z либо символом подчеркивания `_`;
- ◆ после первого символа можно использовать буквы от a до z или от A до Z, цифры от 0 до 9 либо символы подчеркивания;
- ◆ символы в верхнем и нижнем регистрах различаются.

Для того чтобы создать функцию, пишут ее определение. Вот общий формат определения функции в Python:

```
def имя_функции():
 инструкция
 инструкция
```

Первая строка называется заголовком функции. Он отмечает начало определения функции. Заголовок функции начинается с ключевого слова **def**, после которого идет **имя\_функции**, затем круглые скобки и потом двоеточие.

Начиная со следующей строки, идет набор инструкций, который называется блоком. Блок – это просто набор инструкций, которые составляют одно целое. Эти инструкции исполняются всякий раз, когда функция вызывается. В приведенном выше общем формате все инструкции в блоке выделены отступом для того, чтобы интерпретатор Python использовал их для определения начала и конца блока.

Рассмотрим пример определения пользовательской функции, которая не возвращает значение:

```
[52] 1 def error_message():
 2 print('*'*25)
 3 print('* Произошла ошибка! *')
 4 print('*'*25)
```

Но запуск данного кода не приведет к выполнению инструкций в строках 2 – 4, так как это только определение функции **error\_message()**. Для выполнения функции ее необходимо вызвать:

 1 error\_message()

```

* Произошла ошибка! *

```

В данном случае определенная ранее функция вызывается и все инструкции выполняются. Следует обратить внимание, что в рассмотренном выше примере пользовательская функция **error\_message()** несколько раз вызывает встроенную функцию **print**.

Рассмотрим пример, когда одна пользовательская функция вызывает другую пользовательскую функцию необходимое количество раз:

```
[56] 1 def main():
2 for _ in range(3):
3 error_message()
```

```
1 main()
→ *****
* Произошла ошибка!

* Произошла ошибка!

* Произошла ошибка!

```

В первом блоке определяется функция `main()`, во втором блоке эта функция вызывается.

Приступая к написанию программы, вы знаете имена функций, которые планируете использовать, но еще не представляете всех деталей кода, который будет в этих функциях. В этом случае можно использовать ключевое слово `pass` для создания пустых функций. Позже, когда детали кода будут известны, можно вернуться к пустым функциям и заменить ключевое слово `pass` содержательным кодом. Интерпретатор Python игнорирует ключевое слово `pass`, и в результате создаст функции, которые ничего не делают.

Ключевое слово `pass` можно использовать в качестве местозаполнителя в любом месте программного кода Python. Например, его можно использовать в инструкциях `if`, `for`, `while`.

### 4.3. Область действия и локальные переменные

Локальная переменная создается внутри функции. Инструкции, которые находятся за пределами функции, к ней доступа не имеют. Разные функции могут иметь локальные переменные с одинаковыми именами, потому что функции не видят локальные переменные друг друга.

Всякий раз, когда переменной внутри функции присваивается значение, в результате создается локальная переменная. Она принадлежит функции, в которой создается, и к такой переменной могут получать доступ только инструкции в этой функции (термин «локальный» указывает на то обстоятельство, что переменная может использоваться лишь локально внутри функции, в которой она создается).

Область действия переменной – это часть программы, в которой можно обращаться к переменной. Переменная видима только инструкциям в области действия переменной. Областью действия переменной является функция, в которой переменная создается. Никакая инструкция за пределами функции не может обращаться к такой переменной. К локальной переменной не может обращаться программный код, который появляется внутри функции в точке до того, как переменная была создана.

#### **4.4. Передача аргументов в функцию**

Аргумент – это любая порция данных, которая передается в функцию, когда функция вызывается. Параметр – это переменная, которая получает аргумент, переданный в функцию.

Иногда полезно не только вызвать функцию, но и отправить одну или более порций данных в функцию. Порции данных, которые отправляются в функцию, называются аргументами.

Функция может использовать свои аргументы в вычислениях или других операциях. Если требуется, чтобы функция получала аргументы, когда она вызывается, то необходимо оборудовать эту функцию одной или несколькими параметрическими переменными. Параметрическая переменная, часто именуемая просто параметром, – это специальная переменная, которой присваивается значение аргумента, когда функция вызывается. Пример функции с параметрической переменной:

```
▶ 1 def print_my_function(x):
2 print('*'*50)
3 print('Значение x =', x)
4 print('Квадрат x^2 =', x**2)
5 print('Куб x^3 =', x**3)
6 print('Произвольная x^4 + 3*x^2 + 7 =', x**4 + 3*x**2 + 7)
```

Функция `print_my_func(x)` с параметром `x` определена и ее теперь можно использовать, передавая вместо формального параметра `x` фактический параметр:

```
▶ 1 print_my_function(4)
2 print_my_function(14.6)
3 print_my_function(77)

→ *****
Значение x = 4
Квадрат x^2 = 16
Куб x^3 = 64
Произвольная x^4 + 3*x^2 + 7 = 311

Значение x = 14.6
Квадрат x^2 = 213.16
Куб x^3 = 3112.136
Произвольная x^4 + 3*x^2 + 7 = 46083.6656

Значение x = 77
Квадрат x^2 = 5929
Куб x^3 = 456533
Произвольная x^4 + 3*x^2 + 7 = 35170835
```

В функции может быть произвольное количество параметров. В таком случае они указываются в определении функции через запятую:

```
▶ 1 def foo(x, y, z):
2 length = x**2 + y**2 + z**2
3 print(length)
4
5 foo(1, 2, 3)
```

14

В строках 1-3 производится определение функции, в строке 5 – вызов функции с передачей фактических параметров.

## 4.5. Возврат значений из функции

Функция с возвратом значения – это функция, которая возвращает значение обратно в ту часть программы, которая ее вызвала. Функция с возвратом значения – это особый тип функций. Она похожа на функцию без возврата значения следующим образом:

- ◆ это группа инструкций, которая выполняет определенную задачу;
- ◆ когда нужно выполнить функцию, ее вызывают.

Когда функция с возвратом значения завершается, она возвращает значение назад в ту часть программы, которая ее вызвала. Возвращаемое из функции значение используется как любое другое значение: оно может быть присвоено переменной, выведено на экран, использовано в математическом выражении (если оно является числом) и т. д.

Функцию с возвратом значения пишут точно так же, как и функцию без возврата значения, но с одним исключением: функция с возвратом значения должна иметь инструкцию **`return`**.

Вот общий формат определения функции с возвратом значения в Python:

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

Одной из инструкций в функции должна быть инструкция **`return`**, которая принимает приведенную ниже форму:

```
return выражение
```

Значение выражения, которое следует за ключевым словом **`return`**, будет отправлено в ту часть программы, которая вызвала функцию. Это может быть любое значение, переменная либо выражение, которые имеют значение (к примеру, математическое выражение).

Рассмотрим пример, демонстрирующий определение и вызов функций с возвратом и без возврата значений:

```

1 # Функция с возвратом значения
2 def f(x):
3 return x*x
4
5 # Функция без возврата значения
6 def g(x):
7 print('We are in g(). x^3 =', x*x*x)
8
9 f(3)
10 g(4)
11
12 print(f(12))
13 print(g(21))

```

→ We are in g(). x^3 = 64  
144  
We are in g(). x^3 = 9261  
None

В строках 1-7 определяются функции **f** и **g**. Но внутри функции **f** присутствует инструкция **return**, за которой следует значение, которое необходимо вернуть из функции. Зарезервированное слово **return** применяется для немедленного завершения работы функции, выхода из нее с одновременным возвратом значения.

В строке 9 производится вызов **f(3)**, которая выполняется, возвращает значение 9, но возврат нигде не сохраняется. В строке 10 производится вызов **g(4)**. Этот вызов ничего не возвращает, но внутри функции производится печать **'We are in g(). x^3 = 64'**.

В строке 12 производится печать вызова функции **f**, это приводит выполнение функции **f**, возврату из него значения, которое и передается в качестве параметра функции **print**.

В строке 13 производится печать вызова функции **g**, это приводит выполнение функции **g**, внутри функции печатается строка **'We are in g(). x^3 = 9261'**. Но функция **g** ничего не возвращает, поэтому в функцию **print** передать нечего... Тот факт, что ничего не вернулось отражается в Python неопределенным значением **None**.

## 4.6. Функции стандартной библиотеки и инструкция `import`

Python, а также большинство языков программирования поставляются вместе со стандартной библиотекой функций, которые были уже написаны за вас. Эти функции, так называемые библиотечные функции, упрощают работу программиста, потому что с их помощью решаются многие задачи. На самом деле, вы уже применяли несколько библиотечных функций Python. Вот некоторые из них: `print`, `input` и `range`. В Python имеется множество других библиотечных функций.

Некоторые библиотечные функции Python встроены в интерпретатор Python. Если требуется применить в программе одну из таких встроенных функций, нужно просто вызвать эту функцию. Это относится, например, к функциям `print`, `input`, `range` и другим, с которыми вы уже познакомились. Однако многие функции стандартной библиотеки хранятся в файлах, которые называются модулями. Эти модули копируются на компьютер при установке языка Python и помогают систематизировать функции стандартной библиотеки. Например, все функции для выполнения математических операций хранятся вместе в одном модуле, функции для работы с файлами – в другом модуле и т. д.

Для того чтобы вызвать функцию, которая хранится в модуле, нужно вверху программы написать инструкцию импорта `import`. Она сообщает интерпретатору имя модуля, который содержит функцию. Например, один из стандартных модулей Python называется `math`. В нем содержатся различные математические функции, которые работают с числами с плавающей точкой. Если требуется применить в программе какую-либо функцию из модуля `math`, необходимо в начале программы написать инструкцию импорта:

```
import math
```

Эта инструкция приводит к загрузке интерпретатором содержимого математического модуля `math` в оперативную память и в результате все функции модуля `math` становятся доступными в программе.

Поскольку внутреннее устройство библиотечных функций невидимо, многие программисты их рассматривают как черные ящики. Термин «черный ящик» используется для описания любого механизма, который принимает нечто на входе, выполняет с полученным некоторую работу (которую невозможно наблюдать) и производит результат на выходе.

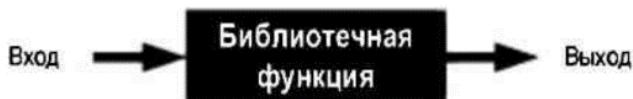


Рисунок 4.2 – Библиотечная функция в виде черного ящика

Рассмотрим пример использования библиотечных функций из модуля `random` и `math`. Модуль `random` позволяет генерировать случайные числа, а модуль `math` содержит математические функции:

```

1 import math, random
2
3 # генерируем 10 случайных чисел
4 # и вычисляем синус от этих значений
5 for _ in range(10):
6 x = random.randint(100, 1000)
7 y = math.sin(x)
8 print('x =', x, '\t y =', y)

```

```

x = 295 y = -0.30478191109030295
x = 937 y = 0.7211003682427743
x = 278 y = 0.99952109184891
x = 121 y = 0.9988152247235795
x = 697 y = -0.42011232727019937
x = 437 y = -0.31325740971087507
x = 190 y = 0.9977992786806003
x = 802 y = -0.7795038357248097
x = 257 y = -0.5733571748155426
x = 524 y = 0.6019757972528995

```

В строке 6 вызывается функция `randint` из импортированного модуля `random` с указанием границ диапазона случайного числа (от 100 до 1000). В строке 7 полученное случайное число передается в качестве параметра функции `sin` из модуля `math`. Следует обратить внимание, что перед именем вызываемой функции указывается модуль ее содержащий. Этого можно избежать, если импортировать не модуль полностью, а только требуемые функции, например, предыдущий код можно переписать следующим образом:

```

1 from math import sin
2 from random import randint
3
4 # генерируем 10 случайных чисел
5 # и вычисляем синус от этих значений
6 for _ in range(10):
7 x = randint(100, 1000)
8 y = sin(x)
9 print('x =', x, '\t y =', y)

x = 142 y = -0.5877950071674065
x = 654 y = 0.5216024406758817
x = 714 y = -0.7568419012610816
x = 619 y = -0.10604746068673451
x = 507 y = -0.9333313464826719
x = 311 y = 0.01767178546737087
x = 773 y = 0.16741513842849595
x = 914 y = 0.20206131455610385
x = 285 y = 0.7738715902084317
x = 306 y = -0.9537617134939987

```

## 4.7. Рекурсия

В программировании существует возможность вызова функции в теле этой же функции, то есть функция вызывает сама себя. Такое явление называется рекурсией, а такая функция – рекурсивной функцией. Синтаксис рекурсивной функции с возвратом значения:

```

def имя_функции(параметры1):
 инструкция1
 инструкция2
 ...
 return имя_функции(параметры2)

```

Рекурсивная функция может и не возвращать значений:

```

def имя_функции(параметры1):
 инструкция1
 инструкция2
 ...
 имя_функции(параметры2)
 инструкция3
 имя_функции(параметры3)
 инструкция4

```

...

Пример рекурсивной функции для вычисления суммы чисел от 1 до N:

```
1 def recursive_sum(n):
2 if n<2:
3 return 1
4 else:
5 return n + recursive_sum(n-1)
```

```
1 print(recursive_sum(20))
```

210

В программировании правило: любая рекурсия может быть представлена в виде цикла (**for, while**).

#### 4.8. Функции в вычислительных задачах

Рассмотрим пример решения задачи с использованием пользовательских функций.

Задач 1. Напишите программу для вычисления площадей прямоугольника, круга, правильного треугольника, объема шара, объема куба. Пользователю должна быть предоставлена возможность выбора решаемой задачи или завершения программы.

Решение. Воспользуемся нисходящей методологией проектирования программы. Для этого выявим отдельные подзадачи в решаемой задаче и оформим подзадачи в виде функций. Подзадачи сведены в таблицу 3.1.

Таблица 3.1 – Подзадачи и функции их реализующие

| Подзадача                                   | Описание                                                             | Функция           |
|---------------------------------------------|----------------------------------------------------------------------|-------------------|
| Вычисление площади прямоугольника           | $S = a \cdot b$ , $a, b$ – стороны прямоугольника                    | s_rectangle(a, b) |
| Вычисление площади круга                    | $S = \pi R^2$ , $R$ – радиус круга                                   | s_circle(r)       |
| Вычисление площади правильного треугольника | $S = \frac{a^2\sqrt{3}}{4}$ , $a$ – сторона правильного треугольника | s_triangle(a)     |

|                        |                                              |            |
|------------------------|----------------------------------------------|------------|
| Вычисление объема шара | $V = \frac{4\pi R^3}{3}$ , $R$ – радиус шара | v_ball(r)  |
| Вычисление объема куба | $V = a^3$ , $a$ – сторона куба               | v_cube(a)  |
| Диалог с пользователем | Организация диалога с пользователем          | question() |

Указанные функции реализуются следующим образом:

```
[15] 1 from math import pi

[6] 1 def s_rectangle(a, b):
2 return a*b

[7] 1 def s_circle(r):
2 return pi*r*r

[8] 1 def s_triangle(a):
2 return a*a*3**0.5 / 4

[9] 1 def v_ball(r):
2 return 4*r**3 / 3

[10] 1 def v_cube(a):
2 return a**3

[11] 1 def question():
2 print('*'*20)
3 print('1 - Вычисление площади прямоугольника')
4 print('2 - Вычисление площади круга')
5 print('3 - Вычисление площади правильного треугольника')
6 print('4 - Вычисление объема шара')
7 print('5 - Вычисление объема куба')
8 answer = input('Выберите команду > ')
9 return answer
```

```

1 while True:
2 choice = question()
3 if choice=='1':
4 a, b = int(input('Введите a > ')), int(input('Введите b > '))
5 print('Площадь прямоугольника: ', s_rectangle(a, b))
6 elif choice=='2':
7 r = int(input('Введите r > '))
8 print('Площадь круга: ', s_circle(r))
9 elif choice=='3':
10 a = int(input('Введите сторону a > '))
11 print('Площадь правильного треугольника: ', s_triangle(a))
12 elif choice=='4':
13 r = int(input('Введите r > '))
14 print('Объем шара: ', v_ball(r))
15 elif choice=='5':
16 a = int(input('Введите a > '))
17 print('Объем куба: ', v_cube(a))
18 else:
19 print('Программа завершена!')
20 break

```

```

1 - Вычисление площади прямоугольника
2 - Вычисление площади круга
3 - Вычисление площади правильного треугольника
4 - Вычисление объема шара
5 - Вычисление объема куба
Выберите команду > 2
Введите r > 4
Площадь прямоугольника: 50.26548245743669

1 - Вычисление площади прямоугольника
2 - Вычисление площади круга
3 - Вычисление площади правильного треугольника
4 - Вычисление объема шара
5 - Вычисление объема куба
Выберите команду > 1
Введите a > 6
Введите b > 7
Площадь прямоугольника: 42

1 - Вычисление площади прямоугольника
2 - Вычисление площади круга
3 - Вычисление площади правильного треугольника
4 - Вычисление объема шара
5 - Вычисление объема куба
Выберите команду > 0
Программа завершена!

```

## Вопросы для самопроверки по теме 4

1. Что такое функция?
2. Что означает фраза «разделяй и властвуй»?

3. Каким образом функции помогают повторно использовать программный код?

4. Каким образом функции ускоряют разработку многочисленных программ?

5. Каким образом функции упрощают разработку программ командами программистов?

6. Из каких двух частей состоит определение функции?

7. Что означает фраза «вызывать функцию»?

8. Что происходит, когда во время исполнения функции достигнут конец ее блока инструкций?

9. Почему необходимо выделять отступом инструкции в блоке?

10. Что такое локальная переменная? Каким образом ограничивается доступ к локальной переменной?

11. Что такое область действия переменной?

12. Разрешается ли, чтобы локальная переменная в одной функции имела одинаковое имя, что и у локальной переменной в другой функции?

13. Чем отличается функция с возвратом значения от функций без возврата значения?

14. Что такое библиотечная функция?

15. Почему библиотечные функции похожи на "черные ящики"?

16. Что делает приведенная ниже инструкция?

`x = random.randint(1, 100)`

17. Что делает приведенная ниже инструкция?

`print(random.randint(1, 20))`

18. Для чего используется следующий вызов библиотечной функции?

`random.seed(20)`

19. Каким образом функции помогают повторно использовать код в программе?

20. Назовите и опишите две части определения функции.

21. Что происходит при исполнении функции, когда достигается конец блока функции?
22. Что такое локальная переменная? Какие инструкции могут обращаться к локальной переменной?
23. Какова область видимости локальной переменной?
24. Почему глобальные переменные затрудняют отладку программы?
25. Предположим, что вы хотите выбрать случайное число из приведенной ниже последовательности: 0, 5, 10, 15, 20, 25, 30. Какую библиотечную функцию вы бы применили?
26. Какую инструкцию вы должны иметь в функции с возвратом значения?
27. Какие три элемента перечислены в таблице «ввод-обработка-вывод»?
28. Что такое булева функция?
29. В чем преимущества от разбиения большой программы на модули?

## Лекция 5. Файлы и исключения

### План лекции

1. Основы файлового ввода-вывода.
2. Запись данных в файл.
3. Чтение данных из файла.
4. Применение циклов для обработки файлов.
5. Обработка исключительных ситуаций.
6. Вычисления с использованием файлов.

### 5.1. Основы файлового ввода-вывода

Когда программе нужно сохранить данные для дальнейшего использования, она пишет эти данные в файл. Позднее их можно прочитать из файла.

Программы, которые мы рассматривали до сих пор, требуют, чтобы пользователь повторно вводил данные при каждом запуске программы, потому что хранящиеся в ОЗУ данные (к которым обращаются переменные) исчезают, как только программа заканчивает свою работу. Если нужно, чтобы программа между своими выполнениями удерживала данные, в ней должна быть предусмотрена возможность их записи. Данные записываются в файл, который обычно хранится на диске компьютера. Сохраненные в файле данные, как правило, остаются в нем после завершения работы программы, и их можно извлечь и использовать в дальнейшем.

Большинство коммерческих пакетов программного обеспечения, используемых ежедневно, хранят данные в файлах.

◆ Текстовые процессоры. Программы обработки текста служат для написания писем, записок, отчетов и других документов. Документы сохраняются в файлах, чтобы их можно было редактировать и распечатывать.

◆ Графические редакторы. Программы редактирования изображений используются для создания графиков и редактирования

изображений, в частности тех, которые вы снимаете цифровой камерой. Создаваемые или редактируемые графическим редактором изображения сохраняются в файлах.

◆ Электронные таблицы. Программы обработки электронных таблиц применяются для работы с числовыми данными. Числа и математические формулы могут вставляться в ячейки электронной таблицы. Затем электронная таблица может быть сохранена в файле для дальнейшего использования.

◆ Игры. Многие компьютерные игры содержат данные в файлах. Например, некоторые игры хранят в файле список имен игроков с их очками. Эти игры, как правило, показывают имена игроков в порядке возрастания количества их очков с наибольшего до наименьшего. Некоторые игры также позволяют сохранять в файле текущее состояние игры, чтобы можно было выйти из игры и потом продолжить игру без необходимости начинать с начала.

◆ Веб-браузеры. Иногда при посещении веб-страницы браузер хранит на компьютере небольшой файл, так называемый файл cookie. Cookie-файлы, как правило, содержат информацию о сеансе просмотра, такую как содержимое корзины с покупками.

Программы, которые используются в ежедневных деловых операциях, в значительной мере опираются на файлы. Программы начисления заработной платы содержат данные о сотрудниках, складские программы содержат данные об изделиях компании, системы бухгалтерского учета – данные о финансовых операциях компании и т. д. – все они хранят свои данные в файлах.

Программисты обычно называют процесс сохранения данных в файле записью данных в файл. Когда часть данных пишется в файл, она копируется из переменной, находящейся в ОЗУ, в файл (рис. 3.1). Термин «файл вывода» используется для файла, в который данные сохраняются. Он имеет такое название, потому что программа помещает в него выходные данные.

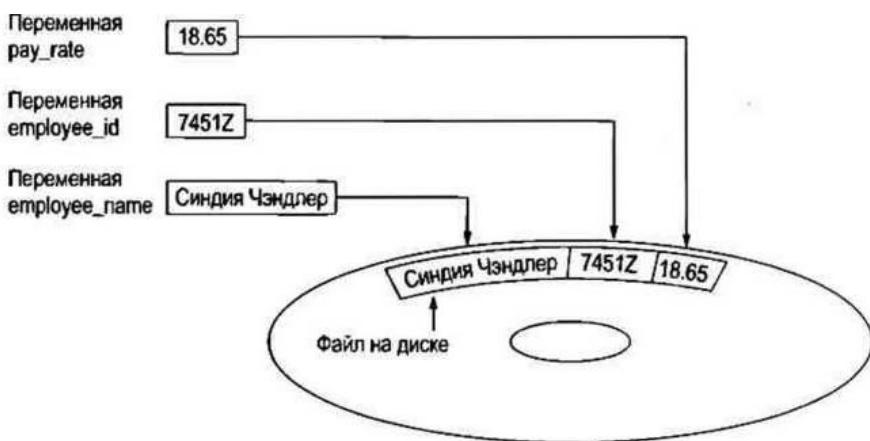


Рисунок 5.1 – Запись данных в файл

Процесс извлечения данных из файла называется чтением данных из фата. Когда порция данных считывается из файла, она копируется из файла в ОЗУ, где на нее ссылается переменная (рис. 5.2). Термин «файл ввода» используется для файла, из которого данныечитываются. Он называется так потому, что программа извлекает входные данные из этого файла.

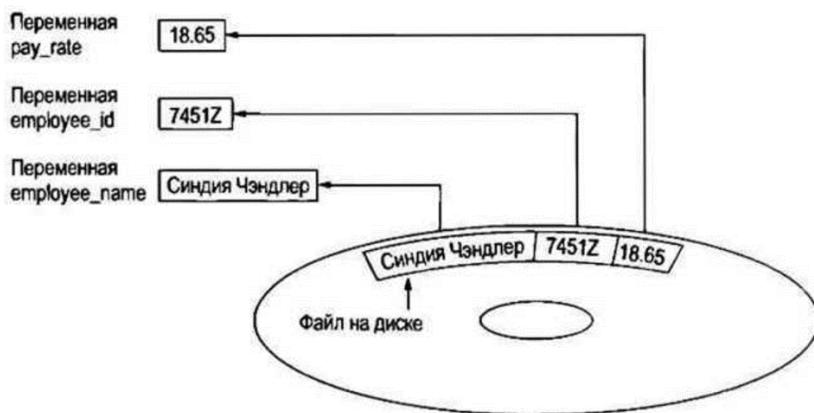


Рисунок 5.2 – Чтение данных из файла

Когда в программе используется файл, всегда требуется выполнить три шага.

1. Открыть файл. В процессе открытия файла создается связь между файлом и программой. Открытие файла вывода обычно создает файл на диске и позволяет программе записать в него данные. Открытие файла ввода позволяет программе прочитать данные из файла.

2. Обработать файл. На этом шаге данные либо записываются в файл (если это файл вывода), либо считаются из файла (если это файл ввода).

3. Закрыть файл. Когда программа закончила использовать файл, его нужно закрыть. Эта операция разрывает связь файла с программой.

### 5.1.1. Типы файлов

Существует два типа файлов: текстовые и двоичные. Текстовый файл содержит данные, которые были закодированы в виде текста при помощи такой схемы кодирования, как ASCII или Юникод. Даже если файл содержит числа, они в файле хранятся как набор символов.

В результате файл можно открыть и просмотреть в текстовом редакторе, таком как Блокнот. Двоичный файл содержит данные, которые не были преобразованы в текст. Данные, которые помещены в двоичный файл, предназначены только для чтения программой, и значит, такой файл невозможно просмотреть в текстовом редакторе.

Несмотря на то что Python позволяет работать и с текстовыми, и с двоичными файлами, в рамках данного курса будем работать только с текстовыми файлами, чтобы можно было использовать текстовый редактор для исследования файлов, создаваемых программами.

### 5.1.2. Методы доступа к файлам

Большинство языков программирования обеспечивает два разных способа получения доступа к данным, хранящимся в файле: последовательный доступ и прямой доступ. Во время работы с файлом с последовательным доступом происходит последовательное обращение к данным, с самого начала файла и до его конца. Если требуется прочитать порцию данных, которая размещена в конце файла, придется прочитать все данные, которые идут перед ней, – перескочить непосредственно к нужным данным не получится.

Во время работы с файлом с прямым доступом (который также называется файлом с произвольным доступом) можно непосредственно перескочить к любой порции данных в файле, не читая данные, которые идут

перед ней. Это подобно тому, как работает проигрыватель компакт-дисков или MP3-плеер. Можно прыгнуть к любой песне, которую нужно прослушать.

В этом курсе будем использовать файлы с последовательным доступом.

### 5.1.3. Имена файлов и файловые объекты

Большинство пользователей компьютеров привыкли к тому, что файлы определяются по их имени. Например, когда вы создаете документ текстовым процессором и сохраняете документ в файле, то вы должны указать имя файла. Когда для исследования содержимого диска вы используете такой инструмент, как Проводник Windows, вы видите список имен файлов.

Каждая операционная система имеет собственные правила именования файлов. Многие системы поддерживают использование расширений файлов, т. е. коротких последовательностей символов, которые расположены в конце имени файла и предваряются точкой. Например, файлы могут иметь расширения jpg, txt и docx. Расширение обычно говорит о типе данных, хранящихся в файле. Например, расширение jpg сообщает о том, что файл содержит графическое изображение, сжатое согласно стандарту изображения JPEG. Расширение txt – о том, что файл содержит текст. Расширение docx (а также расширение doc) – что файл содержит документ Microsoft Word.

Для того чтобы программа работала с файлом, находящимся на диске компьютера, она должна создать в оперативной памяти файловый объект. Файловый объект – это программный объект, который связан с определенным файлом и предоставляет программе методы для работы с этим файлом. В программе на файловый объект ссылается переменная. Она используется для осуществления любых операций, которые выполняются с файлом (рис. 5.3).

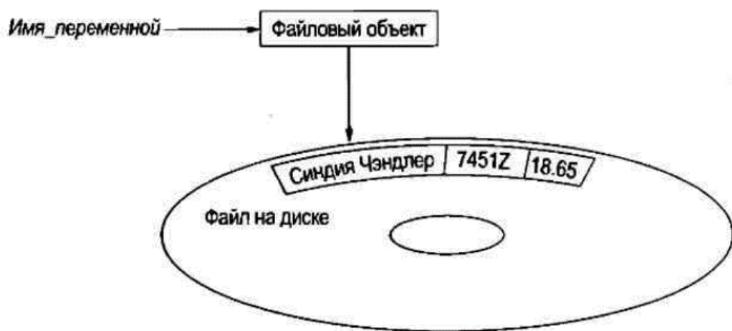


Рисунок 5.3 – Имя переменной ссылается га файловый объект, связанный с файлом

#### 5.1.4. Открытие файла

В Python функция **open** применяется для открытия файла. Она создает файловый объект и связывает его с файлом на диске. Вот общий формат применения функции **open**:

**файловая\_переменная = open(имя\_файла, режим)**

Здесь **файловая\_переменная** – это имя переменной, которая ссылается на файловый объект; **имя\_файла** – это строковый литерал, задающий имя файла; **режим** – это строковый литерал, задающий режим доступа (чтение, запись и т. д.), в котором файл будет открыт.

В табл. 5.1 представлены три строковых литерала, которые можно использовать для задания режима доступа.

Таблица 5.1 – Некоторые режимы доступа к файлам в Python

| Режим | Описание                                                                                                                                            |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| 'r'   | Открыть файл только для чтения. Файл не может быть изменен, в него нельзя записать                                                                  |
| 'w'   | Открыть файл для записи. Если файл уже существует, то стереть его содержимое. Если файл не существует, то создать его                               |
| 'a'   | Открыть файл, в который будет выполнена запись. Все записываемые в файл данные будут добавлены в его конец. Если файл не существует, то создать его |

Например, предположим, что файл **customers.txt** содержит данные о клиентах, и мы хотим его открыть для чтения. Вот пример вызова функции **open**:

```
customer_file = open('customers.txt', 'r')
```

После исполнения этой инструкции будет открыт файл **customers.txt** и переменная **customer\_file** будет ссылаться на файловый объект, который можно использовать для чтения данных из файла.

Предположим, что мы хотим создать файл с именем **sales.txt** и записать в него данные. Вот пример вызова функции **open**:

```
salesfile = open('sales.txt', 'w')
```

После исполнения этой инструкции будет создан файл **sales.txt** и переменная **sales\_file** будет ссылаться на файловый объект, который можно использовать для записи данных в файл.

При использовании режима '**w**' на диске создается файл. Если при открытии файла с указанным именем он уже существует, то содержимое существующего файла будет удалено.

### 5.1.5. Указание места расположения файла

Когда в функцию **open** передается имя файла, которое в качестве аргумента не содержит путь, интерпретатор Python исходит из предположения, что место расположения файла такое же, что и у программы. Например, предположим, что программа расположена на компьютере, работающем под управлением Windows, в папке C:\Users\Documents\Python. Если программа выполняется, и она исполняет инструкцию:

```
test_file = open('test.txt', 'w')
```

то файл test.txt создается в той же папке. Если требуется открыть файл в другом месте расположения, можно указать путь и имя файла в аргументе, который передается в функцию **open**. Если указать путь в строковом литерале (в особенности на компьютере под управлением Windows), следует снабдить строковый литерал префиксом в виде буквы **r**. Вот пример:

```
test_file = open(r'C:\Users\temp\test.txt', 'w')
```

Эта инструкция создает файл test.txt в папке C:\Users\temp. Префикс `r` указывает на то, что строковый литерал является неформатированным. В результате этого интерпретатор Python рассматривает символы обратной косой черты как обычные символы. Без префикса `r` интерпретатор предположит, что символы обратной косой черты являются частью экранированных последовательностей, и произойдет ошибка.

## 5.2. Запись данных в файл

До сих пор в данном курсе мы работали с несколькими библиотечными функциями Python и даже создавали свои функции. Теперь рассмотрим другой тип функций, которые называются методами. Метод – это функция, которая принадлежит объекту и выполняет некоторую операцию с использованием этого объекта. После открытия файла для выполнения операций с файлом используются методы файлового объекта.

Например, файловые объекты имеют метод `write( )`, который применяется для записи данных в файл. Вот общий формат вызова метода `write( )`:

```
файловая_переменная.write(строковое_значение)
```

В данном коде `файловая_переменная` – это переменная, которая ссылается на файловый объект, `строковое_значение` – символьная последовательность, которая будет записана в файл. Файл должен быть открыт для записи (с использованием режима '`w`' или '`a`'), либо произойдет ошибка.

Давайте допустим, что `customer_file` ссылается на файловый объект, и файл открыт для записи в режиме '`w`'. Вот пример записи в файл строкового значения 'Чарльз Пейс':

```
customer_file.write('Чарльз Пейс')
```

Приведенный ниже фрагмент кода демонстрирует еще один пример:

```
name = 'Чарльз Пейс'
customer_file.write(name)
```

Вторая инструкция пишет в файл, связанный с переменной `customer_file`, значение, на которое ссылается переменная `name`. В данном случае она запишет в файл строковое значение 'Чарльз Пейс'.

После того как программа закончила работать с файлом, она должна закрыть его. Это действие разрывает связь программы с файлом. В некоторых системах невыполнение операции закрытия файла вывода может вызвать потерю данных. Это происходит потому, что данные, которые пишутся в файл, сначала пишутся в буфер, т. е. небольшую «область временного хранения» в оперативной памяти. Когда буфер полон, система пишет содержимое буфера в файл. Этот прием увеличивает производительность системы потому, что запись данных в оперативную память быстрее их записи на диск. Процесс закрытия файла вывода записывает любые несохраненные данные, которые остаются в буфере, в файл.

В Python для закрытия файла применяется метод `close( )` файлового объекта. Например, приведенная ниже инструкция закрывает файл, который связан с `customer_file`:

```
customer_file.close()
```

В следующей программе приведен законченный код на Python, который открывает файл вывода, пишет в него данные и затем его закрывает:

```
1 f = open('customers.txt', 'w')
2 f.write('Иванов П.А.\n')
3 f.write('Мирнов В.Г.\n')
4 f.write('Седов Е.В.\n')
5 f.close()
```

### 5.3. Чтение данных из файла

Если файл был открыт для чтения (с помощью режима '`r`'), то для чтения всего его содержимого в оперативную память применяют метод файлового объекта `read( )`. При вызове метода `read( )` он возвращает содержимое файла в качестве строкового значения. Например, в следующей

программе показано применение метода `read()` для чтения содержимого файла `customers.txt`, который мы создали ранее.

```
1 f=open('customers.txt')
2 data = f.read()
3 f.close()
4
5 print(data)
```

Иванов П.А.  
Мирнов В.Г.  
Седов Е.В.

Таким образом, метод `read()` считывает все содержимое файла.

Если требуется считывать содержимое файла построчно, то применяется метод `readline()`. Предположим в файле содержатся данные о продажах (одно число в каждой строке). Содержимое файла:

| money.txt | X       |
|-----------|---------|
| 1         | 1005.89 |
| 2         | 2650    |
| 3         | 9065.43 |

Чтобы вычислить сумму продаж, воспользуемся следующим кодом:

```
1 f=open('money.txt')
2 data1 = float(f.readline())
3 data2 = float(f.readline())
4 data3 = float(f.readline())
5 f.close()
6 total = data1 + data2 + data3
7 total = round(total, 2)
8 print(total)
```

12721.32

## 5.4. Применение циклов для обработки файлов

При чтении файла можно использовать методы `read()` или `readline()`, но какую функцию использовать если необходимо прочитать 100, 1000 или 10 000 строк, в каждой из которых расположены числа, которые необходимо

обработать. В данном случае нельзя просто вызвать `readline()` множеством раз, но можно воспользоваться операторами цикла `for`, `while`.

Создадим файл и запишем в него 1000 числовых значений. Будем записывать случайные числа, для этого воспользуемся методами модуля `random`:

```
▶ 1 from random import random, seed
 2 seed(10)
 3 f = open('money.txt', 'w')
 4 for _ in range(1000):
 5 money = round(random() * 10000, 2)
 6 f.write(str(money) + '\n')
 7 f.close()
 8 print('Task completed!')
```

Task completed!

В результате получим файл `money.txt`, содержащий 1000 вещественных чисел, расположенных построчно:

|    | money.txt × | ... |
|----|-------------|-----|
| 1  | 5714.03     |     |
| 2  | 4288.89     |     |
| 3  | 5780.91     |     |
| 4  | 2060.98     |     |
| 5  | 8133.21     |     |
| 6  | 8235.89     |     |
| 7  | 6534.73     |     |
| 8  | 1602.3      |     |
| 9  | 5206.69     |     |
| 10 | 3277.73     |     |
| 11 | 2499.97     |     |
| 12 | 9528.17     |     |
| 13 | 9965.57     |     |
| 14 | 445.56      |     |
| 15 | 8601.61     |     |
| 16 | 6031.91     |     |
| 17 | 3816.06     |     |
| 18 | 2836.18     |     |
| 19 | 6749.65     |     |

Используя данный файл необходимо решить задачу: вычислить среднее арифметическое всех чисел в файле и определить минимальное число, содержащееся в файле.

Язык Python позволяет писать цикл `for`, автоматически читающий строки в файле без проверки какого-либо особого условия, которое

сигнализирует о конце файла. Этот цикл не требует операции первичного чтения и автоматически останавливается, когда достигнут конец файла.

Вот общий формат такого цикла:

**for переменная in файловый\_объект:**

инструкция

инструкция

В данном формате **переменная** – это имя переменной, **файловый\_объект** – это переменная, которая ссылается на файловый объект. Данный цикл будет выполнять одну итерацию для каждой строки в файле. Во время первой итерации цикла **переменная** будет ссылаться на первую строку в файле (как на символьную последовательность), во время второй итерации она будет ссылаться на вторую строку и т. д.

Решение задачи с использованием подобного цикла показано в следующем листинге:

```
1 f=open('money.txt')
2 min_value = 10**6
3 count = 0
4 sum_value = 0
5 # Обходим файл построчно
6 for line in f:
7 money = float(line)
8 sum_value += money
9 count += 1
10 min_value = min(min_value, money)
11 f.close()
12 print(f'Минимум = {min_value}, Среднее = {sum_value/count}')
13
```

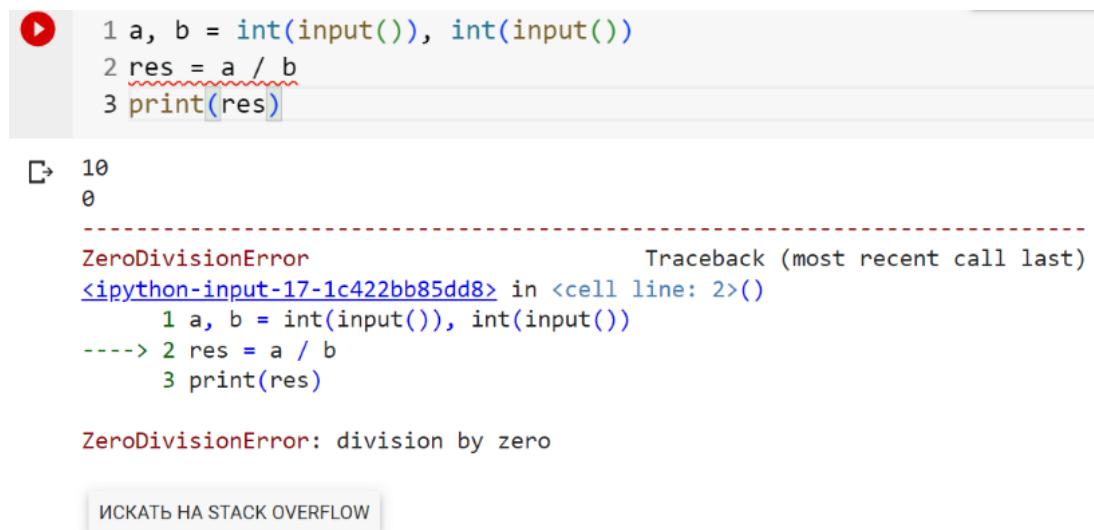
Минимум = 5.57, Среднее = 5033.190079999994

В представленном листинге обход файла производится с использованием цикла **for** подобно обходу некоторой коллекции, то есть файл представлен для Python как последовательность текстовых строк.

## 5.5. Обработка исключительных ситуаций

Исключение – это ошибка, которая происходит во время работы программы, приводящая к ее внезапному останову. Для корректной обработки исключений используется инструкция **try/except**.

Исключение – это ошибка, которая происходит во время работы программы. В большинстве случаев исключение приводит к внезапному останову программы. Например, взгляните следующий листинг. Программа получает от пользователя два числа и делит первое число на второе. Но при выполнении программы произошло исключение, потому что в качестве второго числа пользователь ввел 0 (деление на 0 вызывает исключение, потому что оно математически невозможно).



```

1 a, b = int(input()), int(input())
2 res = a / b
3 print(res)

10
0

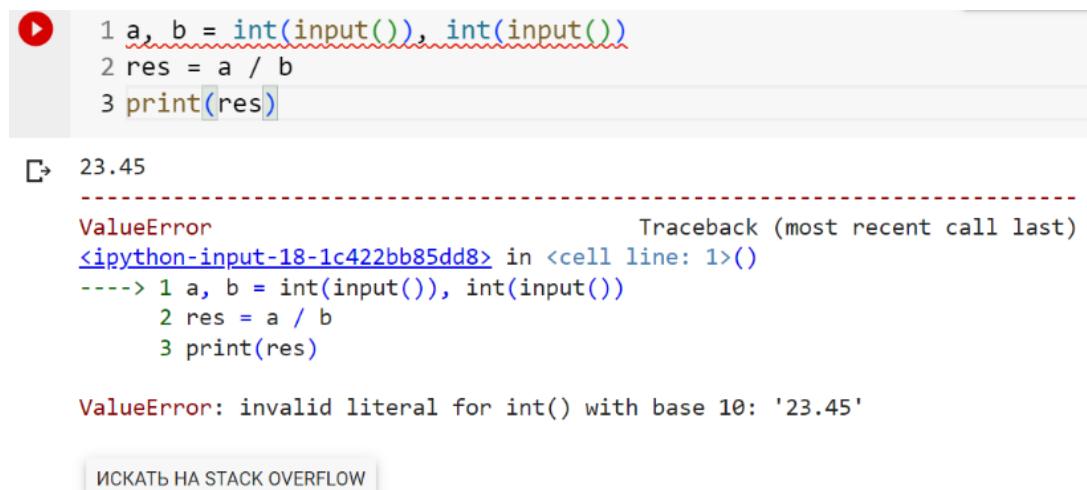
ZeroDivisionError Traceback (most recent call last)
<ipython-input-17-1c422bb85dd8> in <cell line: 2>()
 1 a, b = int(input()), int(input())
----> 2 res = a / b
 3 print(res)

ZeroDivisionError: division by zero

```

[ИСКАТЬ НА STACK OVERFLOW](#)

Допустим, что пользователям объяснили, что в качестве значения  $b$  нельзя вводить 0. Но пользователь может ввести различные значения, которые также могут привести к генерации исключительной ситуации. Например:



```

1 a, b = int(input()), int(input())
2 res = a / b
3 print(res)

23.45

ValueError Traceback (most recent call last)
<ipython-input-18-1c422bb85dd8> in <cell line: 1>()
----> 1 a, b = int(input()), int(input())
 2 res = a / b
 3 print(res)

ValueError: invalid literal for int() with base 10: '23.45'

```

[ИСКАТЬ НА STACK OVERFLOW](#)

В данной программе пользователь ввел первое число, но оно является вещественным, а программа ожидает целой число.

Таким образом существует большое разнообразие возможных действий пользователя, которые могут привести к ошибке в программе, и программа завершится аварийно. Возникновение многих исключений можно предотвратить, тщательно продумывая свою программу. Однако некоторых исключений невозможно избежать независимо от того, насколько тщательно написана программа.

Язык Python, как и большинство современных языков программирования, позволяет писать программный код, который откликается на вызванные исключения и препятствует внезапному аварийному останову программы. Такой программный код называется обработчиком исключений и пишется при помощи инструкции **try/except**. Существует несколько способов написания инструкции **try/except**, но приведенный ниже общий формат показывает самый простой ее вариант:

**try:**

инструкция

инструкция

**except ИмяИсключения:**

инструкция

инструкция

В начале данной инструкции стоит ключевое слово **try**, сопровождаемое двоеточием. Затем идет блок кода, который мы будем называть группой **try**. Она состоит из одной или нескольких инструкций, которые потенциально могут вызвать исключение.

После группы **try** идет выражение **except**. Оно начинается с ключевого слова **except**, за которым необязательно может следовать имя исключения с двоеточием. Начиная с последующей строки, располагается блок инструкций, который мы будем именовать обработчиком.

Во время исполнения инструкции **try/except** начинают исполняться инструкции в группе **try**. Ниже описывается, что происходит далее.

- ◆ Если инструкция в группе **try** вызывает исключение, которое задано в выражении **except ИмяИсключения**, то выполняется обработчик,

который расположен сразу после выражения `except`. Затем программа возобновляет выполнение инструкцией, которая идет сразу после инструкции `try/except`.

- ◆ Если инструкция в группе `try` вызывает исключение, которое не задано в выражении `except ИмяИсключения`, то программа остановится и выведет сообщение об ошибке в отчете об обратной трассировке.
- ◆ Если инструкции в группе `try` выполняются, не вызывая исключений, то любые выражения `except` и обработчики в данной инструкции пропускаются, и программа возобновляет исполнение инструкцией, которая идет сразу после инструкции `try/except`.

Например, вот таким образом можно защитить программу от нулевого значения делителя:

```
▶ 1 try:
 2 a, b = int(input()), int(input())
 3 res = a / b
 4 print(res)
 5 except ZeroDivisionError:
 6 print('Нельзя делить на ноль!')
```

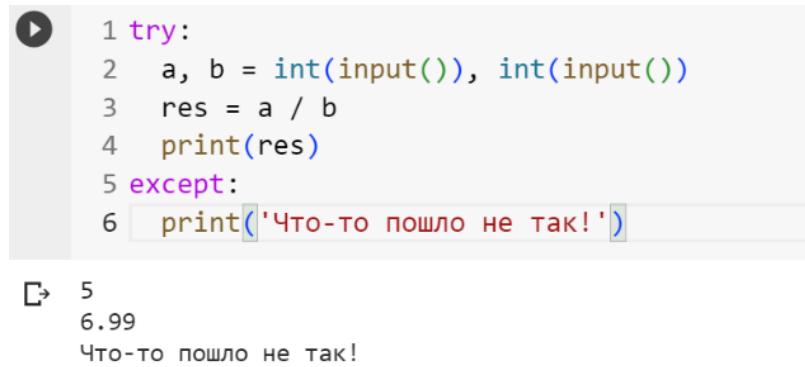
```
⇨ 1
0
Нельзя делить на ноль!
```

Но программа все еще неустойчива к вещественным значениям, эту ситуацию можно исправить так:

```
▶ 1 try:
 2 a, b = int(input()), int(input())
 3 res = a / b
 4 print(res)
 5 except ZeroDivisionError:
 6 print('Нельзя делить на ноль!')
 7 except ValueError:
 8 print('Необходимо вводить только целые числа!')
```

```
45.5
Необходимо вводить только целые числа!
```

Чтобы отловить множества исключений без их детализации по именам можно использовать упрощенный синтаксис:



```

1 try:
2 a, b = int(input()), int(input())
3 res = a / b
4 print(res)
5 except:
6 print('Что-то пошло не так!')

```

5  
6.99  
Что-то пошло не так!

При таком варианте обработки не имеет значение какое именно исключение приведет к краху программы, все равно управление будет передано блоку **except**.

Инструкция **try/except** может иметь необязательное выражение **finally**, которое должно появляться после всех выражений **except**. Вот общий формат инструкции **try/except** с выражением **finally**:

```

try:
 инструкция
 инструкция
except ИмяИсключения:
 инструкция
 инструкция
finally:
 инструкция
 инструкция

```

Блок инструкций, который появляется после выражения **finally**, называется группой **finally**. Инструкции в группе **finally** всегда исполняются после инструкций в группе **try** и после того, как любые обработчики исключений исполнились. Инструкции в группе **finally** исполняются независимо от того, произошло исключение или нет. Цель группы **finally** состоит в том, чтобы выполнять операции очистки, такие как закрытие файлов или других ресурсов. Любой программный код, который написан в группе **finally**, будет всегда исполняться, даже если группа **try** вызовет исключение.

## Вопросы для самопроверки по теме 5

1. Что такое файл вывода?
2. Что такое файл ввода?
3. Какие три шага программа должна сделать, когда она использует файл?
4. Какие бывают два типа файлов? Каково различие между ними?
5. Какие бывают два типа доступа к файлу? Каково различие между ними?
6. С какими двумя именами, связанными с файлами, необходимо работать в своем программном коде при написании программы, которая выполняет файловую операцию?
7. Что происходит с уже существующим файлом при попытке его открыть как файл вывода (используя режим 'w')?
8. Какова задача открытия файла?
9. Какова задача закрытия файла?
10. Что такое позиция считывания файла? Где первоначально находится позиция считывания при открытии файла ввода?
11. В каком режиме открывается файл, если требуется записать в него данные, но при этом требуется оставить существующее содержимое файла нетронутым? В какую часть файла данные записываются при этом?
12. Дайте краткое объяснение, что такое исключение.
13. Что происходит, если вызвано исключение, и программа его не обрабатывает инструкцией `try/except`?
14. Какое исключение программа вызывает, когда она пытается открыть несуществующий файл?
15. Какое исключение программа вызывает, когда она применяет функцию `float( )` для конвертации нечислового строкового значения в число?

16. Опишите три шага, которые должны быть сделаны, когда в программе используется файл.

17. Почему программа должна закрыть файл, когда она закончила его использовать?

18. Что такое позиция считывания файла? Где она находится при открытии файла для чтения?

19. Что происходит с существующим содержимым файла, если существующий файл открывается в режиме дозаписи?

20. Что происходит, если файл не существует и программа пытается его открыть в режиме дозаписи?

## Лекция 6. Списки

### План лекции

1. Изменяемые и неизменяемые последовательности.
2. Список. Индексация.
3. Изменение элементов списка
4. Поиск элементов списка.
5. Операции над списками. Методы списков.

Последовательность – это объект, содержащий многочисленные значения, которые следуют одно за другим. Над последовательностью можно выполнять операции для проверки значений и управления хранящимися в ней значениями.

Последовательность – это объект в виде индексируемой коллекции, который содержит многочисленные значения данных. Хранящиеся в последовательности значения следуют одно за другим. Python предоставляет разнообразные способы выполнения операций над значениями последовательностей.

### **6.1. Изменяемые и неизменяемые последовательности.**

В Python имеется несколько разных типов объектов-последовательностей. В этой теме рассматриваются два фундаментальных типа: списки и кортежи. Списки и кортежи – это последовательности, которые могут содержать разные типы данных. Отличие списков от кортежей простое: список является мутабельной последовательностью (т. е. программа может изменять его содержимое), а кортеж – немутабельной последовательностью (т. е. после его создания его содержимое изменить невозможно). Мы рассмотрим ряд операций, которые можно выполнять над этими последовательностями, включая способы получения доступа и управления их содержимым.

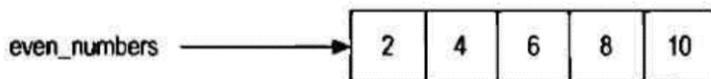
## 6.2. Список. Индексация.

Список – это объект, который содержит многочисленные элементы данных. Списки являются мутабельными последовательностями, т. е. их содержимое может изменяться во время исполнения программы. Списки являются динамическими структурами данных, т. е. элементы в них могут добавляться или удаляться из них. Для работы со списками в программе можно применять индексацию, нарезку и другие разнообразные методы.

Каждая хранящаяся в списке порция данных называется элементом. Ниже приведена инструкция, которая создает список целых чисел:

```
even_numbers = [2, 4, 6, 8, 10]
```

Значения, заключенные в скобки и разделенные запятыми, являются элементами списка. После исполнения приведенной выше инструкции переменная **even\_numbers** будет ссылаться на список:



Вот еще один пример:

```
names = ['Молли', 'Стивен', 'Уилл', 'Алисия', 'Адриана']
```

Эта инструкция создает список из пяти строковых значений. После ее исполнения переменная **names** будет ссылаться на список:



Список может содержать значения разных типов:

```
info = ['Алисия', 27, 1550.87]
```

Эта инструкция создает список, содержащий строковое значение, целое число и число с плавающей точкой. После исполнения инструкции переменная **info** будет ссылаться на список:



Для вывода списка можно использовать функцию **print**:

```
1 numbers = [5, 10, 15, 20]
2 print(numbers)
```

⇨ [5, 10, 15, 20]

Python имеет также встроенную функцию `list( )`, которая преобразует некоторые типы данных в тип списка. Например, из предыдущих лекций известно, что `range( )` порождает некоторый итерируемый объект (некоторую последовательность, которую можно обойти в цикле). Для преобразования итерируемого объекта в список можно использовать функцию `list( )`:

```
1 iter = range(5)
2 print(iter, type(iter))
3 my_list = list(iter)
4 print(my_list, type(my_list))
```

⇨ range(0, 5) <class 'range'>
[0, 1, 2, 3, 4] <class 'list'>

В данном листинге в строке 1 создается итерируемый объект, на который ссылается переменная `iter`, в строке 2 производится печать объекта и информации о его типе. В строке 3 вызывается функция `list( )` для преобразования объекта `iter` в списковый тип данных.

Можно создать список из итерируемого объекта гораздо лаконичнее:

```
1 tens = list(range(10,100, 10))
2 print(tens)
```

[10, 20, 30, 40, 50, 60, 70, 80, 90]

### 6.2.1. Оператор повторения

Символ `*` в Python перемножает два числа. Но когда операнд слева от символа `*` является последовательностью (в частности, списком), а операнд справа – целым числом, он становится оператором повторения. Оператор повторения делает многочисленные копии списка и все их объединяет. Вот общий формат операции:

```
список * n
```

В данном формате **список** – это обрабатываемый список, **n** – число создаваемых копий. Приведенный ниже код это демонстрирует:

```
1 zeros = [0] * 7
2 print(zeros)
```

```
⇒ [0, 0, 0, 0, 0, 0, 0]
```

Рассмотрим каждую инструкцию.

- ◆ В строке 1 выражение `[0] * 7` делает семь копий списка `[0]` и объединяет их в единый список. Получившийся список присваивается переменной **zeros**.
- ◆ В строке 2 переменная **zeros** передается функции **print**. Результат функции выводится.

Вот еще одна демонстрация повторения списка:

```
1 nums = [1, 5, 10] * 3
2 print(nums)
```

```
⇒ [1, 5, 10, 1, 5, 10, 1, 5, 10]
```

Очевидно, что кроме оператора повторения, к списку можно применить оператор объединения `+`:

```
1 nums = [1, 2, 3]
2 all_nums = nums + [4, 5, 6]
3 print(all_nums)
```

```
⇒ [1, 2, 3, 4, 5, 6]
```

### 6.2.2. Последовательный обход списка в цикле for

Один из самых простых способов доступа к отдельным элементам в списке состоит в использовании цикла **for**. Вот общий формат:

**for** переменная **in** список:

инструкция

инструкция

В общем формате **переменная** – это имя переменной, которая будет принимать последовательные значения из списка, а список – это имя списка. Всякий раз, когда цикл повторяется, переменная будет ссылаться на копию элемента в списке, начиная с первого элемента:

```
▶ 1 numbers = [1, 2, 3, 4]
 2 for num in numbers:
 3 print(num)
```

```
◀ 1
 2
 3
 4
```

Во время первой итерации цикла переменная **num** будет ссылаться на значение 1, во время второй итерации цикла переменная **num** будет ссылаться на значение 2 и т. д. Это показано на рис. 6.1.

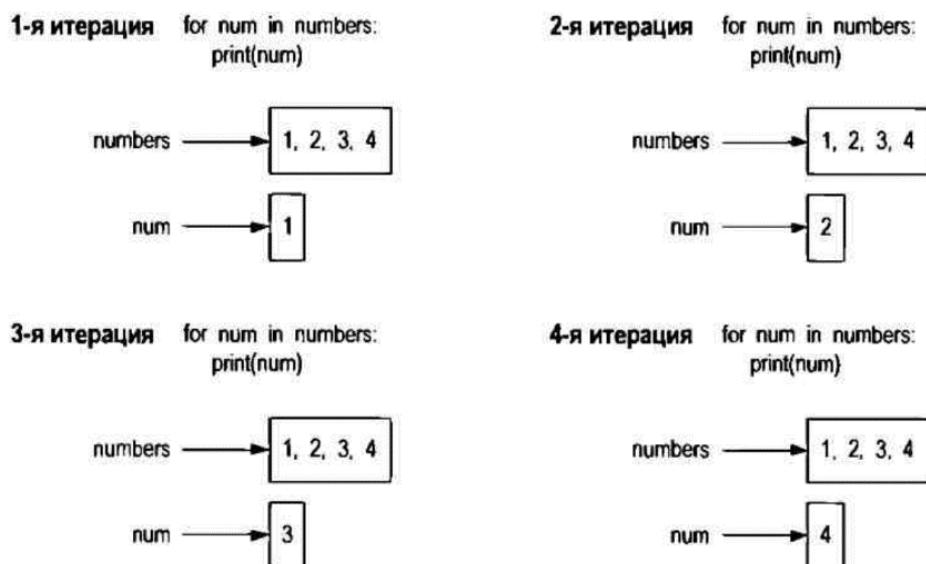


Рисунок 6.1 – Обход списка [1, 2, 3, 4] в цикле

Важно понимать, что мы не можем использовать переменную **num** для изменения содержимого элемента в списке. Если мы изменим значение, на которое ссылается **num**, то это изменение не повлияет на список. Пример кода:

```
▶ 1 numbers = [1, 2, 3, 4]
 2 for num in numbers:
 3 num = 99
 4 print(numbers)
```

```
◀ [1, 2, 3, 4]
```

Следует понимать также, что переменная, указывающая на элемент списка при обходе, может указывать на элементы различного типа:

```
▶ 1 elements = [2, 4, 'Hello', 5.67, [7, 8, 1]]
 2 for elem in elements:
 3 print(elem)

2
4
Hello
5.67
[7, 8, 1]
```

### 6.2.3. Индексация

Еще один способ доступа к отдельным элементам в списке реализован посредством индексации. Каждый элемент в списке имеет индекс, который определяет его позицию в списке.

Индексация начинается с 0, поэтому индекс первого элемента равняется 0, индекс второго элемента равняется 1 и т. д. Индекс последнего элемента в списке на 1 меньше числа его элементов.

Например, приведенная ниже инструкция создает список с четырьмя элементами:

```
my_list = [10, 20, 30, 40]
```

Индексы элементов в этом списке будут 0, 1, 2 и 3. Напечатать элементы списка можно при помощи инструкции:

```
print(my_list[0], my_list[1], my_list[2], my_list[3])
```

Приведенный ниже цикл тоже напечатает элементы списка:

```
▶ 1 my_list = [10, 20, 30, 40]
 2 index = 0
 3 while index < 4:
 4 print(my_list[index])
 5 index += 1

⇨ 10
20
30
40
```

Для идентификации позиций элементов относительно конца списка можно также использовать отрицательные индексы. Для того чтобы определить позицию элемента, интерпретатор Python прибавляет отрицательные индексы к длине списка. Индекс -1 идентифицирует последний элемент списка, -2 – предпоследний элемент и т. д. Например:

```
 1 my_list = [10, 20, 30, 40]
 2 print(my_list[-1], my_list[-2], my_list[-3], my_list[-4])
 ↵ 40 30 20 10
```

При обращении к элементу по несуществующему индексу возникнет ошибка:

```
 1 my_list = [10, 20, 30, 40]
 2 index = 0
 3 while index < 5:
 4 print(my_list[index])
 5 index += 1
 ↵ 10
 20
 30
 40

IndexError Traceback (most recent call last)
 in <cell line: 3>()
 2 index = 0
 3 while index < 5:
----> 4 print(my_list[index])
 5 index += 1

IndexError: list index out of range
```

Когда этот цикл начнет последнюю итерацию, переменной `index` будет присвоено значение 4, которое является недопустимым индексом списка. В результате инструкция, которая вызывает функцию `print`, вызовет исключение `IndexError`.

#### 6.2.4. Функция `len`

Python имеет встроенную функцию `len`, которая возвращает длину последовательности, в частности, списка. Вот пример:

```
▶ 1 my_list = [10, 20, 30, 40]
 2 size = len(my_list)
 3 print(size)
```

⇨ 4

Первая инструкция присваивает переменной `my_list` список `[10, 20, 30, 40]`. Вторая инструкция вызывает функцию `len`, передавая переменную `my_list` в качестве аргумента. Эта функция возвращает значение 4, т. е. количество элементов в списке, которое присваивается переменной `size`.

Функция `len` может применяться для предотвращения исключения `IndexError` во время последовательного обхода списка в цикле. Вот пример:

```
▶ 1 my_list = [10, 20, 30, 40]
 2 index = 0
 3 while index < len(my_list):
 4 print(my_list[index])
 5 index += 1
```

10  
20  
30  
40

### 6.3. Изменение элементов списка

Списки в Python являются мутабельными последовательностями, т. е. их элементы могут изменяться. Следовательно, выражение в форме `список[индекс]` может появляться слева от оператора присваивания. Например:

```
▶ 1 numbers = [1, 2, 3, 4, 5]
 2 print(numbers)
 3 numbers[0] = 99
 4 print(numbers)
```

⇨ [1, 2, 3, 4, 5]
 [99, 2, 3, 4, 5]

Инструкция в строке 3 присваивает 99 элементу `numbers[0]`. Она изменяет первое значение в списке на 99.

Когда для присвоения значения элементу списка применяется выражение индексации, следует использовать допустимый индекс существующего элемента, в противном случае произойдет исключение **IndexError**. Например:

```
▶ 1 numbers = [1, 2, 3, 4, 5]
 2 numbers[10] = 99

→ -----
IndexError Traceback (most recent call last)
<ipython-input-16-106f552b03bc> in <cell line: 2>()
 1 numbers = [1, 2, 3, 4, 5]
----> 2 numbers[10] = 99

IndexError: list assignment index out of range
```

[ИСКАТЬ НА STACK OVERFLOW](#)

Список чисел, который создается в первой инструкции, имеет пять элементов с индексами от 0 до 4. Вторая инструкция вызовет исключение **IndexError**, потому что список чисел не содержит элемент с индексом 10.

Рассмотрим, пример, в котором элементы списка вводятся пользователем с клавиатуры:

```
▶ 1 list_size = 5
 2 my_list = [0] * list_size
 3
 4 for i in range(list_size):
 5 my_list[i] = float(input(f'Введите {i}-ый элемент > '))
 6
 7 print(my_list)

→ Введите 0-ый элемент > 45
Введите 1-ый элемент > 56.78
Введите 2-ый элемент > 11.33
Введите 3-ый элемент > 12
Введите 4-ый элемент > 78895545.788856
[45.0, 56.78, 11.33, 12.0, 78895545.788856]
```

## Конкатенирование списков

Конкатенировать означает соединять две части воедино. Для конкатенирования двух списков используется оператор **+**. Вот пример:

```

1 list1 = [1, 2, 3, 4]
2 list2 = [5, 6, 7, 8]
3 list3 = list1 + list2
4 print(list3)

[1, 2, 3, 4, 5, 6, 7, 8]

```

После исполнения этого фрагмента кода списки `list1` и `list2` останутся неизменными.

## 6.4. Поиск элементов списка.

### 6.4.1. Срезы

Выражение среза извлекает диапазон элементов из последовательности. Индексация позволяет извлекать конкретный элемент из последовательности. Иногда возникает необходимость извлечь из последовательности более одного элемента. В Python можно составлять выражения, которые извлекают части последовательности, которые называются срезами.

Срез – это диапазон элементов, которые извлекаются из последовательности. Для того чтобы получить срез списка, пишут выражение в приведенном ниже общем формате:

`имя_списка[начало : конец]`

В данном формате начало – это индекс первого элемента в срезе, конец – индекс, отмечающий конец среза. Это выражение возвращает список, содержащий копию элементов с **начала** до **конца** (но не включая последний). Пример:

```

1 days = ['Понедельник', 'Вторник', 'Среда', \
2 'Четверг', 'Пятница', 'Суббота', 'Воскресенье']
3 mid_days = days[2:5]
4 print(mid_days)

['Среда', 'Четверг', 'Пятница']

```

Примеры получения срезов:

```

1 nums = list(range(1, 21))
2 print(nums)
3 print(nums[5:10])
4 print(nums[:10])
5 print(nums[12:])
6 print(nums[1:15:3])
7 print(nums[-5:])

```

⇨ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
[6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[13, 14, 15, 16, 17, 18, 19, 20]
[2, 5, 8, 11, 14]
[16, 17, 18, 19, 20]

Из представленного примера можно сделать следующие выводы:

1. При взятии среза можно указывать третий параметр – он обозначает шаг среза (строка 6).
2. Можно пропустить указание границ среза (строки 4, 5, 7).
3. В срезах можно использовать отрицательные индексы (строка 7).

#### 6.4.2. Поиск значений в списках при помощи инструкции `in`

Поиск значения в списке выполняется при помощи оператора `in`.

В Python оператор `in` применяется для определения, содержится ли значение в списке. Вот общий формат выражения с оператором `in` для поиска значения в списке:

**значение in список**

В данном формате **значение** – это искомое значение, **список** – список, в котором выполняется поиск. Это выражение возвращает истину, если значение в списке найдено, и ложь в противном случае. Пример:

```

1 prod_nums = ['v331', 'x17', 'k665', 'w5678', 'z2']
2 prod = input('введите код изделия > ')
3
4 if prod in prod_nums:
5 print('Изделие присутствует в списке')
6 else:
7 print('Изделие отсутствует в списке')

```

введите код изделия > d567  
Изделие отсутствует в списке

## 6.5. Операции над списками. Методы списков

Списки имеют многочисленные методы, которые позволяют работать с содержащимися в них значениями. Python также предлагает несколько встроенных функций, которые широко используются для работы со списками (таблица 6.1).

Таблица 6.1 – Методы списков

| Метод                           | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>append(значение)</b>         | Добавляет значение в конец списка                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>index(значение)</b>          | Возвращает индекс первого элемента, значение которого равняется значению. Если значение в списке не найдено, вызывается исключение ValueError                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>insert(индекс, значение)</b> | Вставляет значение в заданную индексную позицию в списке. Когда значение вставляется в список, список расширяется, чтобы разместить новое значение. Значение, которое ранее находилось в заданной индексной позиции, и все элементы после него сдвигаются на одну позицию к концу списка. Если указан недопустимый индекс, исключение не происходит. Если задан индекс за пределами конца списка, значение будет добавлено в конец списка. Если применен отрицательный индекс, который указывает на недопустимую позицию, значение будет вставлено в начало списка |
| <b>sort( )</b>                  | Сортирует значения в списке, в результате чего они появляются в возрастающем порядке (с наименьшего значения до наибольшего)                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>remove(значение)</b>         | Удаляет из списка первое появление значения. Если значение в списке не найдено, вызывается исключение ValueError                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>reverse()</b>                | Инвертирует порядок следования значений в списке, т. е. меняет его на противоположное                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>min(список)</b>              | Возвращает минимальное значение в списке <b>список</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>max(список)</b>              | Возвращает максимальное значение в списке <b>список</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>sum(список)</b>              | Возвращает сумму элементов в списке <b>список</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

Рассмотрим пример использования некоторых методов списков:

```

1 from random import randint, seed
2 seed(10)
3 my_list=[]
4 print('init')
5 for _ in range(10):
6 my_list.append(randint(10, 150))
7 print(my_list)
8
9 print('insert 1000 at 0')
10 my_list.insert(0, 1000)
11 print(my_list)

12
13 print('reverse')
14 my_list.reverse()
15 print(my_list)
16
17 print('remove 135')
18 my_list.remove(135)
19 print(my_list)
20
21 print('min: ', min(my_list))
22 print('max: ', max(my_list))
23 print('sum: ', sum(my_list))
24 print('sort')
25 my_list.sort()
26 print(my_list)

init
[18, 119, 133, 13, 62, 128, 135, 81, 51, 18]
insert 1000 at 0
[1000, 18, 119, 133, 13, 62, 128, 135, 81, 51, 18]
reverse
[18, 51, 81, 135, 128, 62, 13, 133, 119, 18, 1000]
remove 135
[18, 51, 81, 128, 62, 13, 133, 119, 18, 1000]
min: 13
max: 1000
sum: 1623
sort
[13, 18, 18, 51, 62, 81, 119, 128, 133, 1000]

```

## Вопросы для самопроверки по теме 6

1. Что покажет приведенный ниже фрагмент кода?

```

numbers = [1, 2, 3, 4, 5]
numbers[2] = 99
print(numbers)

```

2. Что покажет приведенный ниже фрагмент кода?

```
numbers = list(range(3))
print(numbers)
```

3. Что покажет приведенный ниже фрагмент кода?

```
numbers = [10] * 5
print(numbers)
```

4. Что покажет приведенный ниже фрагмент кода?

```
numbers = list(range(1, 10, 2))
for n in numbers:
 print(n)
```

5. Что покажет приведенный ниже фрагмент кода?

```
numbers = [1, 2, 3, 4, 5]
print(numbers[-2])
```

6. Как найти количество элементов в списке?

7. Что покажет приведенный ниже фрагмент кода?

```
Numbers1 = [1, 2, 3]
numbers2 = [10, 20, 30]
numbers3 = numbers1 + numbers2
print(numbers1)
print(numbers2)
print(numbers3)
```

8. Что покажет приведенный ниже фрагмент кода?

```
numbers1 = (1, 2, 3)
numbers2 = (10, 20, 30)
numbers2 += numbers1
print(numbers1)
```

```
print(numbers2)
```

9. Что покажет приведенный ниже фрагмент кода?

```
numbers = [1, 2, 3, 4, 5]
mylist = numbers[1:3]
print(my_list)
```

10. Что покажет приведенный ниже фрагмент кода?

```
numbers = [1, 2, 3, 4, 5]
my_list = numbers[1:]
print(my_list)
```

11. Что покажет приведенный ниже фрагмент кода?

```
numbers = [1, 2, 3, 4, 5]
my_list = numbers[:1]
print(my_list)
```

12. Что покажет приведенный ниже фрагмент кода?

```
numbers = [1, 2, 3, 4, 5]
my_list = numbers[:]
print(my_list)
```

13. Что покажет приведенный ниже фрагмент кода?

```
numbers = [1, 2, 3, 4, 5]
my_list = numbers[-3:]
print(my_list)
```

14. Как найти минимальное и максимальное значения в списке?

15. Допустим, что в программе появляется следующая инструкция:

```
names = []
```

Какую из приведенных ниже инструкций следует применить для добавления в список в индексную позицию 0 строкового значения 'Вэнди'? Почему вы выбрали именно эту инструкцию вместо другой?

- a) `names(0) = 'Вэнди';`
- б) `names.append('Вэнди').`

16. Опишите приведенные ниже списковые методы:

- a) `index( );`
- б) `insert( );`
- в) `sort( );`
- г) `reverse( )`

## Лекция 7. Строки

### План лекции

1. Основы работы со строками
2. Строковые методы.
3. Чтение csv-файла.

### 7.1. Основы работы со строками

#### 7.1.1. Обход строкового значения в цикле `for`

Python предлагает большое разнообразие инструментов и методов программирования, которые можно использовать для проверки и управления строковыми данными. В сущности, строковые данные представляют собой один из видов последовательности, и поэтому многие подходы, которые применяются к обработке списков, применимы и к строковым данным.

Один из самых простых способов получить доступ к отдельным символам в строковом значении состоит в применении цикла for. Вот общий формат:

```
for переменная in строковое_значение:
 инструкция
 инструкция
```

В данном формате `переменная` – это имя переменной, `строковое_значение` – строковый литерал либо переменная, которые ссылаются на строковое значение. Во время каждой итерации цикла переменная будет ссылаться на копию символа в строковом значении, начиная с первого символа. Говорят, что цикл выполняет последовательный перебор символов в строковом значении. Вот пример:

```
1 name='John'
2 for c in name:
3 print(c)
```

```
J
o
h
n
```

По аналогии со списками, при таком обходе строки, символы в строке поменять нельзя.

### 7.1.2. Индексация

Еще один способ получить доступ к отдельным символам в строковом значении реализуется при помощи индекса. Каждый символ в строковом значении имеет индекс, который задает его позицию. Индексация начинается с 0, поэтому индекс первого символа равняется 0, индекс второго символа равняется 1 и т. д. Индекс последнего символа в строковом значении равняется количеству символов в строковом значении минус 1. Индекс можно использовать для получения копии отдельного символа в строковом значении:

```
1 name="Nikolaev Evgeny"
2 ch = name[0] # Доступ к 0-ому символу
3 print(ch)
4 chs = name[:8] # Получение среза
5 print(chs)
```

```
N
Nikolaev
```

При обращении к символу по несуществующему индексу как и в случае со списками произойдет ошибка **IndexError**.

К строкам применима функция **len( )**, операции срезов и конкатенации **+**, которая выполняет склеивание строк.

В Python данные строкового типа являются немутабельными последовательностями, т. е. после создания их нельзя изменить. Пример:

```

1 name="Nikolaev Evgeny"
2 name[9] = 'e'

→ -----
TypeError Traceback (most recent call last)
<ipython-input-62-8243216000c1> in <cell line: 2>()
 1 name="Nikolaev Evgeny"
----> 2 name[9] = 'e'

TypeError: 'str' object does not support item assignment

```

### 7.1.3. Проверка строковых значений при помощи in и not in

В Python оператор **in** используется с целью определения, содержится ли одно строковое значение в другом. Вот общий формат выражения с использованием оператора **in** с двумя строковыми значениями:

**строковое\_значение1 in строковое\_значение2**

**строковое\_значение1** и **строковое\_значение2** могут быть либо строковыми литералами, либо переменными, которые ссылаются на строковые значения. Это выражение возвращает истину, если **строковое\_значение1** найдено в **строковом значении2**. Пример:

```

1 text = "Восемьдесят семь лет назад"
2 if 'семь' in text:
3 print('Строковое значение "семь" найдено.')
4 else:
5 print('Строковое значение "семь" не найдено.')

```

→ Строковое значение "семь" найдено.

Этот фрагмент кода определяет, содержит ли строковое значение '**Восемьдесят семь лет назад**' подстроку '**семь**'.

## 7.2. Строковые методы

В предыдущих темах было рассмотрено, что метод – это функция, которая принадлежит объекту и выполняет некоторую операцию с использованием этого объекта. Объекты строкового типа в Python имеют

многочисленные методы. В этой теме мы рассмотрим несколько строковых методов для выполнения следующих типов операций:

- ◆ проверка строковых значений;
- ◆ выполнение различных модификаций;
- ◆ поиск подстрок и замена последовательностей символов.

### 7.2.1. Методы проверки строковых значений

Строковые методы, перечисленные в табл. 7.1, проверяют строковое значение в отношении определенных свойств. Например, метод **isdigit()** возвращает истину, если строковое значение содержит только цифры. В противном случае он возвращает ложь. Пример:

```
 1 string1 = '1200'
2 if string1.isdigit():
3 print(f'{string1} содержит только цифры.')
4 else:
5 print(f'{string1} содержит символы, отличные от цифр.')
```

1200 содержит только цифры.

Таблица 7.1 – Методы строк для проверки значений

| Метод            | Описание                                                                                                                                                                                                                                             |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>isalnum()</b> | Возвращает истину, если строковое значение содержит только буквы алфавита или цифры и имеет по крайней мере один символ. В противном случае возвращает ложь                                                                                          |
| <b>isalpha()</b> | Возвращает истину, если строковое значение содержит только буквы алфавита и имеет по крайней мере один символ. В противном случае возвращает ложь                                                                                                    |
| <b>isdigit()</b> | Возвращает истину, если строковое значение содержит только цифры и имеет по крайней мере один символ. В противном случае возвращает ложь                                                                                                             |
| <b>islower()</b> | Возвращает истину, если все буквы алфавита в строковом значении находятся в нижнем регистре, и строковая последовательность содержит по крайней мере одну букву алфавита. В противном случае возвращает ложь                                         |
| <b>isspace()</b> | Возвращает истину, если строковое значение содержит только пробельные символы и имеет по крайней мере один символ. В противном случае возвращает ложь. (Пробельными символами являются пробелы, символы новой строки (\n) и символы табуляции (\t).) |
| <b>isupper()</b> | Возвращает истину, если все буквы алфавита в строковом значении находятся в верхнем регистре, и строковая последовательность содержит по крайней мере одну букву алфавита. В противном случае возвращает ложь                                        |

### 7.2.2. Методы модификации строковых значений

Несмотря на то что строковые данные являются немутуируемыми последовательностями, т. е. их нельзя изменить, они имеют много методов, которые возвращают их видоизмененные версии. В табл. 7.2 представлено несколько таких методов.

Таблица 7.2 – Методы модификации строкового значения

| Метод                 | Описание                                                                                                                                                                                                                                          |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>lower()</b>        | Возвращает копию строкового значения, в котором все буквы преобразованы в нижний регистр. Любой символ, который уже находится в нижнем регистре или не является буквой алфавита, остается без изменения                                           |
| <b>lstrip()</b>       | Возвращает копию строкового значения, в котором все ведущие пробельные символы удалены. Ведущими пробельными символами являются пробелы, символы новой строки (\n) и символы табуляции (\t), которые появляются в начале строкового значения      |
| <b>lstrip(символ)</b> | Аргументом символ является строковое значение, содержащее символ. Возвращает копию строкового значения, в котором удалены все экземпляры символа, появляющиеся в начале строкового значения                                                       |
| <b>rstrip()</b>       | Возвращает копию строкового значения, в котором все замыкающие пробельные символы удалены. Замыкающими пробельными символами являются пробелы, символы новой строки (\n) и символы табуляции (\t), которые появляются в конце строкового значения |
| <b>rstrip(символ)</b> | Аргументом символ является строковое значение, содержащее символ. Возвращает копию строковой последовательности, в которой удалены все экземпляры символа, появляющиеся в конце строкового значения                                               |
| <b>strip()</b>        | Возвращает копию строкового значения, в котором удалены все ведущие и замыкающие пробельные символы                                                                                                                                               |
| <b>strip(символ)</b>  | Возвращает копию строкового значения, в котором удалены все экземпляры символа, появляющиеся в начале и конце строкового значения                                                                                                                 |
| <b>upper()</b>        | Возвращает копию строкового значения, в котором все буквы преобразованы в верхний регистр. Любой символ, который уже находится в верхнем регистре или не является буквой алфавита, остается без изменения                                         |

Пример:

```

1 name = 'Nikolaev Evgeny'
2 name_upper = name.upper()
3 print(name_upper)

```

⇒ NIKOLAEV EVGENY

### 7.2.3. Поиск и замена

Программам очень часто требуется выполнять поиск подстрок или строковых данных, которые появляются внутри других строковых данных. Например, предположим, что вы открыли документ в своем текстовом редакторе, и вам нужно отыскать слово, которое где-то в нем находится. Искомое вами слово является подстрокой, которая появляется внутри более крупной последовательности символов, т. е. документе.

В табл. 7.3 перечислены некоторые строковые методы Python, которые выполняют поиск подстрок, а также метод, который заменяет найденные подстроки другой подстрокой.

Таблица 7.3 – Методы поиска и замены

| Метод                               | Описание                                                                                                                                                                    |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>endswith(подстрока)</code>    | Аргумент подстрока – это строковое значение. Метод возвращает истину, если строковое значение заканчивается подстрокой                                                      |
| <code>find(подстрока)</code>        | Аргумент подстрока – это строковое значение. Метод возвращает наименьший индекс в строковом значении, где найдена подстрока. Если подстрока не найдена, метод возвращает -1 |
| <code>replace(старое, новое)</code> | Аргументы старое и новое – это строковые значения. Метод возвращает копию строкового значения, в котором все экземпляры старых подстрок заменены новыми подстроками         |
| <code>startswith(подстрока)</code>  | Аргумент подстрока – это строковое значение. Метод возвращает истину, если строковое значение начинается с подстроки                                                        |
| <code>count(подстрока)</code>       | Подсчитывает количество вхождений <b>подстрока</b> в текущей строке                                                                                                         |

Пример:

```

1 filename = input('Введите имя файла: ')
2 if filename.endswith('.txt'):
3 print('Это имя текстового файла.')
4 elif filename.endswith('.py'):
5 print('Это имя исходного файла Python.')
6 elif filename.endswith('.doc'):
7 print('Это имя документа текстового редактора.')
8 else:
9 print('Неизвестный тип файла.')
10

```

⇨ Введите имя файла: zuba.doc  
Это имя документа текстового редактора.

Метод **find( )** отыскивает заданную подстроку в строковом значении и возвращает наименьшую индексную позицию подстроки, если она найдена. Если подстрока не найдена, метод возвращает **-1**. Пример:

```

1 string = "Восемьдесят семь лет назад"
2 position = string.find('семь')
3 if position != -1:
4 print(f'Слово "семь" найдено в индексной позиции {position}.')
5 else:
6 print('Слово "семь" не найдено.')

```

⇨ Слово "семь" найдено в индексной позиции 2.

#### 7.2.4. Разбиение строкового значения

Строковые значения в Python имеют метод **split( )**, который возвращает список, содержащий слова в строковом значении. В программе приведен пример:

```

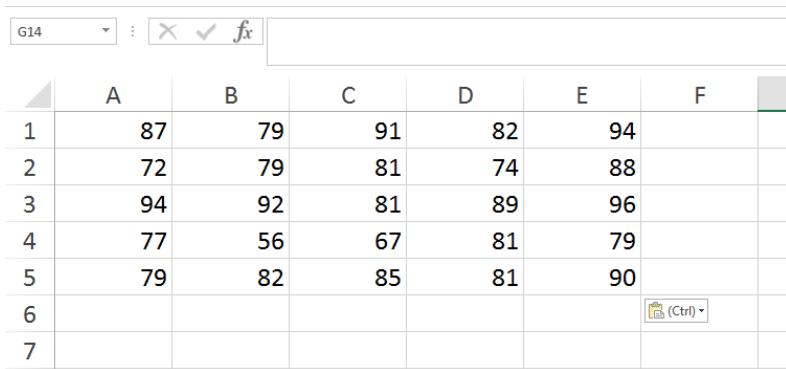
1 my_string = 'Один два три четыре'
2 print(my_string)
3 # Разбить строковое значение.
4 word_list = my_string.split()
5
6 # Напечатать список слов.
7 print(word_list)

```

⇨ Один два три четыре  
['Один', 'два', 'три', 'четыре']

### 7.3. Чтение CSV-файлов

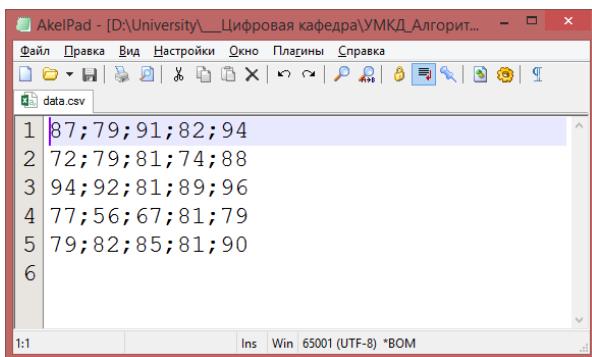
Большинство приложений по работе с электронными таблицами и базами данных могут экспорттировать данные в формат файла CSV (Comma Separated Values – значения, разделенные запятыми). Каждая строка в CSV-файле содержит строковый литерал с элементами данных, разделенными запятыми. Например, предположим, что преподаватель хранит результаты контрольных работ своих учеников в электронной таблице:



|   | A  | B  | C  | D  | E  | F      |
|---|----|----|----|----|----|--------|
| 1 | 87 | 79 | 91 | 82 | 94 |        |
| 2 | 72 | 79 | 81 | 74 | 88 |        |
| 3 | 94 | 92 | 81 | 89 | 96 |        |
| 4 | 77 | 56 | 67 | 81 | 79 |        |
| 5 | 79 | 82 | 85 | 81 | 90 |        |
| 6 |    |    |    |    |    | (Ctrl) |
| 7 |    |    |    |    |    |        |

В каждой строке таблицы содержатся результаты тестов для одного ученика, и у каждого ученика есть пять результатов контрольных работ. Необходимо вычислить средний балл для каждого ученика, вывести средние баллы и вычислить максимальные средний балл.

Python способен работать с файлами Excel, но для этого необходимо подключить библиотеки. В данной теме обработаем данные из Excel предварительно сохранив их в формате CSV (Меню Файл → Сохранить Как...). В результате получим файл вида:



Полученный csv-файл является текстовым, открывается и редактируется текстовыми редакторами и, соответственно, Python также способен обрабатывать текстовые файлы:

```

1 f = open('data.csv')
2 lines = f.readlines()
3 print(lines)
4
5 avg_marks = [] # средние оценки
6
7 for line in lines:
8 marks = line.split(';')
9 print(marks)
10 total = 0
11
12 # Вычисляем сумму баллов
13 for mark in marks:
14 total += float(mark)
15 # Добавляем среднее значение
16 avg_marks.append(total / 5)

17
18 print('Средние баллы: ')
19 print(avg_marks)
20 print(f'Максимальный средний балл: {max(avg_marks)}')

```

Вывод для данного скрипта:

```

['87;79;91;82;94\n', '72;79;81;74;88\n', '94;92;81;89;96\n', '77;56;67;81;79\n', '79;82;85;81;90\n']
['87', '79', '91', '82', '94\n']
['72', '79', '81', '74', '88\n']
['94', '92', '81', '89', '96\n']
['77', '56', '67', '81', '79\n']
['79', '82', '85', '81', '90\n']
Средние баллы:
[86.6, 78.8, 90.4, 72.0, 83.4]
Максимальный средний балл: 90.4

```

### Вопросы для самопроверки по теме 7

1. Допустим, что переменная `name` ссылается на строковое значение. Напишите цикл `for`, который напечатает каждый символ в строковом значении.
2. Каков индекс первого символа в строковом значении?
3. Каков индекс последнего символа в строковом значении, которое имеет 10 символов?

4. Что произойдет, если попытаться применить для доступа к символу в строковом значении недопустимый индекс?

5. Как найти длину строкового значения?

6. Что не так с приведенным ниже фрагментом кода?

```
animal = 'Тигр'
animal(0) = 'Л'
```

7. Что покажет приведенный ниже фрагмент кода?

```
mystring = 'abcdefg'
print(mystring[2:5])
```

8. Что покажет приведенный ниже фрагмент кода?

```
mystring = 'abcdefg'
print(mystring[3:])
```

9. Что покажет приведенный ниже фрагмент кода?

```
mystring = 'abcdefg'
print(mystring[:3])
```

10. Что покажет приведенный ниже фрагмент кода?

```
mystring = 'abcdefg'
print(mystring[:])
```

11. Напишите фрагмент кода с использованием оператора `in`, который определяет, является ли '`d`' подстрокой переменной `mystring`.

12. Допустим, что переменная `big` ссылается на строковое значение. Напишите инструкцию, которая преобразует строковое значение, на которое она ссылается, в нижний регистр и присваивает преобразованный результат переменной `little`.

13. Напишите инструкцию, которая выводит сообщение "Цифра", если строковое значение, на которое ссылается переменная **ch**, содержит цифру. В противном случае эта инструкция должна показать сообщение "Цифр нет".

14. Что покажет приведенный ниже фрагмент кода?

```
ch = 'a'
ch2 = ch.upper()
print(ch, ch2)
```

15. Напишите цикл, который запрашивает у пользователя "Желаете повторить программу или выйти? (П/В)". Цикл должен повторяться до тех пор, пока пользователь не введет В (в верхнем или нижнем регистре).

16. Что покажет приведенный ниже фрагмент кода?

```
var = '$'
print(var.upper())
```

17. Напишите цикл, который подсчитывает количество символов в верхнем регистре, появляющихся в строковом значении, на которое ссылается переменная **mystring**.

18. Допустим, что в программе имеется приведенная ниже инструкция:

```
days = 'Понедельник Вторник Среда'
```

Напишите инструкцию, которая разбивает строковое значение, создавая приведенный ниже список:

```
['Понедельник', 'Вторник', 'Среда']
```

19. Допустим, что в программе имеется приведенная ниже инструкция:

```
values = 'один$два$три$четыре'
```

Напишите инструкцию, которая разбивает строковое значение, создавая приведенный ниже список:

[ 'один', 'два', 'три', 'четыре' ]

## Лекция 8. Коллекции

### План лекции

1. Кортежи.
2. Словари.
3. Множества.
4. СерIALIZАЦИЯ объектов.

### 8.1. Кортежи

В следующем листинге представлен пример работы с кортежем:

```
▶ 1 k = (10, 12, 'Novak B.M.', 3.44)
 2 print(k)
 3 print(type(k))

⇒ (10, 12, 'Novak B.M.', 3.44)
<class 'tuple'>
```

В следующем коде представлен пример работы со списком:

```
▶ 1 k = [10, 12, 'Novak B.M.', 3.44]
 2 print(k)
 3 print(type(k))

[10, 12, 'Novak B.M.', 3.44]
<class 'list'>
```

Представленные листинги отличаются только видом скобок в строке 1 и типом переменной `k`. Набор значений в скобках ( ) Python понимает как кортеж (tuple). Отличия кортежа и списка намного глубже, чем просто различие скобок... Кортеж – это немутуируемая (неизменяемая) коллекция, то есть нельзя поменять элементы кортежа, удалить отдельный элемент, добавить элемент. Но можно получить заданный элемент кортежа, обойти кортеж в цикле:

```
1 k = (10, 12, 'Novak B.M.', 3.44)
2 print(k[2])
3 for elem in k:
4 print(elem)
```

```
Novak B.M.
10
12
Novak B.M.
3.44
```

Кортежи применяются для хранения небольших наборов данных, которые не предполагается изменять в программе. Например, хранение координат точек на плоскости или в пространстве. Скорость работы программы при использовании кортежей выше, чем при использовании списков.

## 8.2. Словари

Словарь – это объект-контейнер, который хранит коллекцию данных. Каждый элемент в словаре имеет две части: ключ и значение. Ключ используют, чтобы установить местонахождение конкретного значения.

В Python словарь – это объект, который хранит коллекцию данных. Каждый хранящийся в словаре элемент имеет две части: ключ и значение. На практике элементы словаря обычно называются парами **"ключ : значение"**. Когда требуется получить из словаря конкретное значение, используется ключ, который связан с этим значением.

Пары **"ключ : значение"** часто называются отображениями, потому что каждому ключу поставлено в соответствие значение, т. е. каждый ключ как бы отображается на соответствующее ему значение.

### 8.2.1. Создание словаря

Словарь создается путем заключения его элементов в фигурные скобки **{ }**. Элемент состоит из ключа, затем двоеточия, после которого идет

значение. Элементы словаря отделяются друг от друга запятыми. Приведенная ниже инструкция демонстрирует пример определения словаря:

```
phonebook = {'Крис': '555-1111',
 'Кэти': '555-2222',
 'Джоанна': '555-3333'
 }
```

Эта инструкция создает словарь и присваивает его переменной **phonebook** (телефонная книга). Словарь содержит три приведенных элемента.

- ◆ Первый элемент – 'Крис' : '555-1111'. В этом элементе ключом является 'Крис', а значением – '555-1111'.
- ◆ Второй элемент – 'Кэти' : '555-2222'. В этом элементе ключом является 'Кэти', а значением – '555-2222'.
- ◆ Третий элемент – 'Джоанна' : '555-3333'. В этом элементе ключом является 'Джоанна', а значением – '555-3333'.

В данном примере ключами и значениями являются строковые объекты. Значения в словаре могут быть объектами любого типа, но ключи должны быть идентифицируемыми объектами. Например, ключами могут быть строковые значения, целые числа, значения с плавающей точкой или кортежи. Ключами не могут быть списки либо мутабельные объекты других типов.

### 8.2.2. Получение значений из словаря

Элементы в словаре не хранятся в каком-то конкретном порядке. Словари не являются последовательностями, как списки, кортежи и строковые значения. Как результат, невозможно использовать числовой индекс для получения значения по его позиции в словаре.

Вместо этого для получения значения используется ключ. Для того чтобы получить значение из словаря, просто пишут выражение в приведенном ниже общем формате:

**имя\_словаря [ключ]**

В данном формате **имя\_словаря** – это переменная, которая ссылается на словарь, **ключ** – это применяемый ключ. Если **ключ** в словаре существует, то выражение возвращает связанное с этим ключом значение. Если ключ не существует, то вызывается исключение **KeyError** (ошибка ключа):

```
1 phonebook = { 'Крис': '555-1111', \
2 'Кэти': '555-2222', \
3 'Джоанна': '555-3333' }
4 print(phonebook['Крис'])
5 print(phonebook['Джоанна'])
6 print(phonebook['Евгений'])
```

→ 555-1111  
555-3333

---

```
KeyError Traceback (most recent call last)
<ipython-input-94-efee0ccb5b63> in <cell line: 6>()
 4 print(phonebook['Крис'])
 5 print(phonebook['Джоанна'])
----> 6 print(phonebook['Евгений'])

KeyError: 'Евгений'
```

Значения для ключей '**Крис**' и '**Джоанна**' были выведены, но для ключа '**Евгений**' возникла ошибка **KeyError**.

### 8.2.3. Применение операторов **in** и **not in** для проверки на наличие значения в словаре

Как продемонстрировано ранее, исключение **KeyError** вызывается при попытке получить из словаря значение с использованием несуществующего ключа. Для того чтобы предотвратить это исключение, можно применить оператор **in**, который определит наличие ключа перед попыткой его использовать для получения значения. Пример проверки:

```
1 phonebook = { 'Крис': '555-1111', \
2 'Кэти': '555-2222', \
3 'Джоанна': '555-3333' }
4 keys = ['Кэти', 'Евгений', 'джоанна']
5 for k in keys:
6 if k in phonebook:
7 print(f'Имя {k}, телефон: {phonebook[k]}')
8 else:
9 print(f"Ключ '{k}' отсутствует")
```

→ Имя Кэти, телефон: 555-2222  
Ключ 'Евгений' отсутствует  
Ключ 'джоанна' отсутствует

Следует иметь в виду, что сравнения строковых значений при помощи операторов **in** и **not in** регистрочувствительны.

#### 8.2.4. Добавление, удаление и изменение значений в словаре

Словари являются мутабельными объектами. В словарь можно добавлять новые пары "ключ : значение", используя для этого инструкцию присваивания в приведенном ниже общем формате:

**имя\_словаря[ключ] = значение**

Здесь **имя\_словаря** – это переменная, которая ссылается на словарь, **ключ** – это применяемый ключ. Если ключ уже в словаре существует, то присвоенное ему значение будет заменено значением. Если же ключ отсутствует, то он будет добавлен в словарь вместе со связанным с ним значением. Пример:

```
▶ 1 pbook = { 'Крис':'555-1111',\
2 'Кэти':'555-2222',\
3 'Джоанна':'555-3333' }
4 pbook['Евгений']='776-3732'
5 print(pbook)
6 pbook['Кэти']='000-1111'
7 print(pbook)

⇨ {'Крис': '555-1111', 'Кэти': '555-2222', 'Джоанна': '555-3333', 'Евгений': '776-3732'}
⇨ {'Крис': '555-1111', 'Кэти': '000-1111', 'Джоанна': '555-3333', 'Евгений': '776-3732'}
```

Удалить элемент из словаря можно с помощью **del**:

```
▶ 1 pbook = { 'Крис':'555-1111',\
2 'Кэти':'555-2222',\
3 'Джоанна':'555-3333' }
4 del pbook['Крис']
5 print(pbook)

⇨ {'Кэти': '555-2222', 'Джоанна': '555-3333'}
```

Получить количество значений элементов в словаре можно с использованием функции **len( )**.

#### 8.2.5. Словарные методы

В таблице 8.1 представлены некоторые методы словаря.

Таблица 8.1 – Методы словаря

| Метод            | Описание                                                                                                                                                           |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>clear()</b>   | Очищает содержимое словаря                                                                                                                                         |
| <b>get()</b>     | Получает значение, связанное с заданным ключом. Если ключ не найден, этот метод не вызывает исключение. Вместо этого он возвращает значение по умолчанию           |
| <b>items()</b>   | Возвращает все ключи в словаре и связанные с ними значения в виде последовательности кортежей                                                                      |
| <b>keys()</b>    | Возвращает все ключи в словаре в виде последовательности кортежей                                                                                                  |
| <b>pop()</b>     | Возвращает из словаря значение, связанное с заданным ключом и удаляет эту пару "ключ : значение". Если ключ не найден, этот метод возвращает значение по умолчанию |
| <b>popitem()</b> | Возвращает в виде кортежа последнюю добавленную в словарь пару "ключ : значение". Этот метод также удаляет пару "ключ : значение" из словаря                       |
| <b>values()</b>  | Возвращает все значения из словаря в виде последовательности кортежей                                                                                              |

### 8.3. Множества

Множество – это объект-контейнер уникальных значений, который работает как математическое множество.

Вот несколько важных моментов, которые следует знать о множествах:

- ◆ Все элементы в множестве должны быть уникальными. Никакие два элемента не могут иметь одинаковое значение.
- ◆ Множества не упорядочены, т. е. элементы в множестве не хранятся в каком-то определенном порядке.
- ◆ Хранящиеся в множестве элементы могут иметь разные типы данных.

Для того чтобы создать множество, необходимо вызвать встроенную функцию **set**. Вот пример создания пустого множества:

```
myset = set()
```

После исполнения этой инструкции переменная **myset** будет ссылаться на пустое множество. В функцию **set** можно также передать один аргумент. Передаваемый аргумент должен быть объектом, который содержит итерируемые элементы, такие как список, кортеж или строковое значение.

Отдельные элементы объекта, передаваемого в качестве аргумента, становятся элементами множества. Вот пример:

```
myset = set(['а', 'б', 'в'])
```

В этом примере в функцию **set** в качестве аргумента передается список. После исполнения этой инструкции переменная **myset** ссылается на множество, содержащее элементы 'а', 'б' и 'в'.

Если в качестве аргумента в функцию **set** передать строковое значение, то каждый отдельный символ в строковом значении становится членом множества. Вот пример:

```
myset = set('абв')
```

После исполнения этой инструкции переменная **myset** будет ссылаться на множество, содержащее элементы 'а', 'б' и 'в'. Это связано с тем фактом, что строка сама по себе является перечислимым типом.

Множества не могут содержать повторяющиеся элементы. Если в функцию **set** передать аргумент, содержащий повторяющиеся элементы, то в множестве появится только один из этих повторяющихся элементов. Вот пример:

```
myset = set('ааабв')
```

Символ 'а' встречается в строковом значении многократно, но в множестве он появится только один раз. После исполнения этой инструкции переменная **myset** будет ссылаться на множество, содержащее три элемента 'а', 'б' и 'в'.

А как быть, если нужно создать множество, в котором каждый элемент является строковым значением, содержащим более одного символа? Например, как создать множество с элементами 'один', 'два' и 'три'? Приведенный ниже фрагмент кода эту задачу не выполнит, потому что в функцию **set** можно передавать не более одного аргумента:

```
Это ОШИБКА!
myset = set('один', 'два', 'три')
```

Верный код для выполнения данной задачи:

```
myset = set(['один', 'два', 'три'])
```

### 8.3.1. Добавление и удаление элементов

Множества являются мутабельными объектами, поэтому элементы можно в них добавлять и удалять из них. Для добавления элемента в множество используется метод `add()`. Приведенный ниже код это демонстрирует:

```
1 s = set()
2 s.add(10)
3 s.add(12)
4 print(s)
```

↙ {10, 12}

Добавление элементов множества можно произвести с использованием метода `update()`:

```
1 s = set([1, 2, 3])
2 s.update([1, 3, 4, 5])
3 print(s)
```

↙ {1, 2, 3, 4, 5}

Элемент из множества можно удалить либо методом `remove()`, либо методом `discard()`. Удаляемый элемент передается в качестве аргумента в один из этих методов, и этот элемент удаляется из множества. Единственная разница между этими двумя методами состоит в том, как они себя ведут, когда указанный элемент в множестве не найден. Метод `remove()` вызывает исключение `KeyError`, а метод `discard()` исключение не вызывает.

### 8.3.2. Применение цикла `for` для обхода множества

Для последовательного перебора всех элементов в множестве цикл `for` используется в приведенном ниже общем формате:

`for переменная in множество:`

инструкция

инструкция

Пример:

```
1 myset=set(range(1, 11, 2))
2 print(myset)
3 for i in myset:
4 print(i)
```

```
1 {1, 3, 5, 7, 9}
2
3
4
5
6
7
8
9
```

Множества также поддерживают операторы **in** и **not in**.

### 8.3.3. Операции над множествами

Множества в Python, подобно математическим множествам поддерживают операции объединения, пересечения, разности и симметричной разности.

Пример вычисления объединения множеств:

```
1 u1 = set('123')
2 u2 = set('678')
3 u3 = u1.union(u2)
4 print(u3)
```

```
1 {'2', '3', '7', '6', '8', '1'}
```

Пример вычисления пересечения множеств:

```
1 u1 = set('12345678')
2 u2 = set('67890')
3 u3 = u1.intersection(u2)
4 print(u3)
```

```
1 {'8', '7', '6'}
```

Пример вычисления разности и симметричной разности множеств:

```
[16] 1 u1 = set('12345678')
2 u2 = set('67890')
3 u3 = u1.difference(u2)
4 print(u3)
```

```
1 {'3', '4', '1', '2', '5'}
```

```

1 u1 = set('12345678')
2 u2 = set('67890')
3 u3 = u1.symmetric_difference(u2)
4 print(u3)

{'3', '0', '1', '9', '2', '4', '5'}

```

## 8.4. Сериализация объектов

Сериализация объекта – это процесс преобразования объекта в поток байтов, которые могут быть сохранены в файле для последующего извлечения. В Python сериализация объекта называется консервацией.

В предыдущих темах было рассмотрено сохранение данных в текстовом файле. Иногда возникает необходимость сохранить в файл содержимое сложного объекта, такого как словарь или множество.

Самый простой способ сохранить объект в файле состоит в сериализации объекта. Когда объект сериализуется, он преобразуется в поток байтов, которые могут быть легко сохранены в файле для последующего извлечения.

В Python процесс сериализации объекта называется консервацией. Стандартная библиотека Python предоставляет модуль **pickle** (маринад), который располагает различными функциями для сериализации, или консервации, объектов. После того как модуль **pickle** импортирован, для консервации объекта выполняются следующие шаги:

1. Открывается файл для двоичной записи.
2. Вызывается метод **dump( )** модуля **pickle**, чтобы законсервировать объект и записать его в указанный файл.
3. После консервации всех объектов, которые требуется сохранить в файл, этот файл закрывается.

Рассмотрим эти шаги подробнее. Для того чтобы открыть файл для двоичной записи, в качестве режима доступа к файлу при вызове функции **open** используется строковый литерал '**wb**'. Например, приведенная ниже инструкция открывает файл с именем **mydata.dat** для двоичной записи:

```
outputfile = open('mydata.dat', 'wb')
```

После открытия файла для двоичной записи вызывается функция **dump** модуля **pickle**. Вот общий формат функции **dump**:

```
pickle.dump(объект, файл)
```

В данном формате объект – это переменная, которая ссылается на объект, подлежащий консервации, файл – это переменная, которая ссылается на файловый объект. После выполнения этой функции программный объект, на который ссылается объект, будет сериализован и записан в файл. Можно законсервировать объект практически любого типа, в том числе списки, кортежи, словари, множества, строковые значения, целые числа и числа с плавающей точкой.

В файл можно сохранить столько законсервированных объектов, сколько необходимо. По завершении работы вызывается метод **close()** файлового объекта для закрытия файла. Пример консервации словаря:

```
1 import pickle
2 phonebook = {'Крис' : '555-1111', \
3 'Кэти' : '555-2222', \
4 'Джоанна' : '555-3333'}
5 output_file = open ('phonebook.dat', 'wb')
6 pickle.dump(phonebook, output_file)
7 output_file.close()
```

Рассмотрим этот скрипт подробнее:

- ◆ Стока 1 импортирует модуль **pickle**.
- ◆ Строки 2-4 создают словарь, содержащий имена (в качестве ключей) и телефонные номера (в качестве значений).
- ◆ Стока 5 открывает файл с именем **phonebook.dat** для двоичной записи.
- ◆ Стока 6 вызывает функцию **dump** модуля **pickle**, чтобы сериализовать словарь **phonebook** и записать его в файл **phonebook.dat**.
- ◆ Стока 7 закрывает файл **phonebook.dat**.

Однажды потребуется извлечь и расконсервировать объекты, которые были законсервированы ранее. Вот шаги, которые выполняются с этой целью:

1. Открывается файл для двоичного чтения.
2. Вызывается функция `load` модуля `pickle`, чтобы извлечь объект из файла и его расконсервировать.
3. После расконсервации из файла всех требующихся объектов этот файл закрывается.

Пример:

```
 1 import pickle
2 input_file = open ('phonebook.dat', 'rb')
3 pb = pickle.load(input_file)
4 input_file.close()
5
6 print(pb)
```

```
⇨ {'Крис': '555-1111', 'Кэти': '555-2222', 'Джоанна': '555-3333'}
```

Рассмотрим этот код подробнее:

- ◆ Стока 1 импортирует модуль `pickle`.
- ◆ Стока 2 открывает файл с именем `phonebook.dat` для двоичного чтения.
- ◆ Стока 3 вызывает функцию `load` модуля `pickle`, чтобы извлечь и расконсервировать объект из файла `phonebook.dat`. Полученный объект присваивается переменной `pb`.
- ◆ Стока 4 показывает словарь, на который ссылается переменная `pb`. Результат выводится в строке 5.
- ◆ Стока 6 закрывает файл `phonebook.dat`.

Очевидно, что можно консервировать и расконсервировать любое количество списков, кортежей, словарей.

### Вопросы для самопроверки по теме 8

1. Элемент в словаре имеет две части. Как они называются?
2. Какая часть элемента словаря должна быть немутабельной?
3. Предположим, что '`старт`': `1472` является элементом словаря. Что является ключом? И что является значением?

4. Предположим, что создан словарь с именем `employee`. Что делает приведенная ниже инструкция?

```
employee['id'] = 54321
```

5. Что покажет приведенный ниже фрагмент кода?

```
stuff = {1:'aaa', 2:'666', 3: 'vvv'}
print(stuff[3])
```

6. Как определить, существует ли пара "ключ : значение" в словаре?

7. Предположим, что существует словарь `inventory`. Что делает приведенная ниже инструкция?

```
del inventory[654]
```

8. Что покажет приведенный ниже фрагмент кода?

```
stuff = {1:'aaa', 2:'666', 3:'vvv'}
print(len(stuff))
```

9. Что покажет приведенный ниже фрагмент кода?

```
stuff = {1:'aaa', 2:'666', 3:'vvv'}
for k in stuff:
 print(k)
```

10. В чем разница между словарными методами `pop()` и `popitem()` применительно к словарям?

11. Что возвращает метод `items()`?

12. Что возвращает метод `keys()`?

13. Что возвращает метод `values()`?

15. Какими являются элементы кортежа: упорядоченными или неупорядоченными? Кортеж является мутируемым типом данных или нет?

16. Какими являются элементы множества: упорядоченными или неупорядоченными?

17. Позволяет ли множество хранить повторяющиеся элементы?
18. Как создать пустое множество?
19. Какие элементы будут храниться в множестве `myset` после исполнения приведенной ниже инструкции?

```
myset = set{'Юпитер'}
```

20. Какие элементы будут храниться в множестве `myset` после исполнения приведенной ниже инструкции?

```
myset = set(25)
```

21. Какие элементы будут храниться в множестве `myset` после исполнения приведенной ниже инструкции?

```
myset = set('ъъъ эээ ююю яяя')
```

22. Какие элементы будут храниться в множестве `myset` после исполнения приведенной ниже инструкции?

```
myset = set([1, 2, 2, 3, 4, 4, 4])
```

23. Какие элементы будут храниться в множестве `myset` после исполнения приведенной ниже инструкции?

```
myset = set(['ттт', 'ффф', 'ююю', 'яяя'])
```

24. Как определяется количество элементов в множестве?

25. Какие элементы будут храниться в множестве `myset` после исполнения приведенной ниже инструкции?

```
myset = set([10, 9, 8])
myset.update([1, 2, 3])
```

26. Какие элементы будут храниться в множестве `myset` после исполнения приведенной ниже инструкции?

```
myset = set([10, 9, 8])
```

```
myset.update('абв')
```

27. В чем разница между методами `discard()` и `remove()`?
28. Как определить, принадлежит ли определенный элемент множеству?
29. Какие элементы будут храниться в множестве `set3` после исполнения приведенной ниже инструкции?

```
set1 = set([10, 20, 30])
set2 = set([100, 200, 300])
set3 = set1.union(set2)
```

30. Какие элементы будут храниться в множестве `set3` после исполнения приведенной ниже инструкции?

```
set1 = set([1, 2, 3, 4])
set2 = set([3, 4, 5, 6])
set3 = set1.intersection(set2)
```

31. Какие элементы будут храниться в множестве `set3` после исполнения приведенной ниже инструкции?

```
set1 = set([1, 2, 3, 4])
set2 = set([3, 4, 5, 6])
set3 = set1.difference(set2)
```

32. Какие элементы будут храниться в множестве `set3` после исполнения приведенной ниже инструкции?

```
set1 = set([1, 2, 3, 4])
set2 = set([3, 4, 5, 6])
set3 = set2.difference(set1)
```

33. Какие элементы будут храниться в множестве `set3` после исполнения приведенной ниже инструкции?

```
set1 = set(['а', 'б', 'в'])
```

```
set2 = set(['б', 'в', 'г'])
set3 = set1.symmetric_difference(set2)
```

34. Взгляните на приведенный ниже фрагмент кода:

```
set1 = set([1, 2, 3, 4])
set2 = set([2, 3])
```

Какое множество является подмножеством другого?

Какое множество является надмножеством другого?

35. Что такое сериализация объекта?

36. Какой режим доступа к файлу используется, когда файл открывается с целью сохранения в нем законсервированного объекта?

37. Какой режим доступа к файлу используется, когда файл открывается с целью извлечения из него законсервированного объекта?

38. Какой модуль следует импортировать, если требуется законсервировать объекты?

39. Какая функция вызывается для консервации объекта?

40. Какая функция вызывается для извлечения и расконсервации объекта?

## Лекция 9. Многомодульные программы

### План лекции

1. Модули и клиенты.
2. Стандартная библиотека Python. Установка библиотек.
3. Модуль os.
4. Модуль math.
5. Модуль itertools.
6. Модуль random.
7. Пользовательские модули.

### 9.1. Модули и клиенты

Модульное программирование – метод создания программного обеспечения, который подчёркивает разделение функциональности программы на независимые взаимозаменяемые блоки, называемые модулями.

Модуль – функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом. Модули проектируются таким образом, чтобы предоставлять программистам удобную для многократного использования функциональность в виде набора функций, классов, констант. Модули могут объединяться в пакеты и, далее, в библиотеки. Удобство использования модульной архитектуры заключается в возможности обновления (замены) модуля, без необходимости изменения остальной системы.

Модульное программирование тесно связано со структурным и объектно-ориентированным программированием, все они имеют одну и ту же цель – облегчить построение больших программ путём декомпозиции на более мелкие части. Все они возникли примерно в 1960-х годах. В настоящее время эти понятия имеют немного разный смысл: модульное программирование теперь относится к высокоуровневой декомпозиции кода всей программы на части; структурное программирование – к низкоуровневому синтезу

программы из структурированных блоков; объектно-ориентированное программирование – к высокоуровневому синтезу программы из объектов, включающих как функции/методы, так и сами данные.

Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Модульность часто является средством упрощения задачи проектирования программы и распределения процесса разработки между группами разработчиков. При разбиении программы на модули для каждого модуля указывается реализуемая им функциональность, а также связи с другими модулями.

Следующие языки поддерживают парадигму модульного программирования (список неполный): Ada, Lisp, D, Erlang, F#, Fortran, Go, Haskell, MATLAB, ML (Standard ML и OCaml), Modula (1, 2, 3), Oberon (1, 2), Pascal (практически все версии), Perl, Python (2 и 3), Ruby, Rust. Во многих других языках также возможно написание программы в виде нескольких отдельных файлов, например, в C, C++, Java, C# и ряде других, однако используемый в них подход нельзя полностью назвать модульным, поскольку отсутствует либо возможность раздельной компиляции модулей, либо возможность писать в модуле произвольные локальные и глобальные переменные (подход Java, когда файл – это класс), в том числе разделять переменные и функции на доступные и недоступные внешним программам (в Java и её наследниках это реализуется за счёт классов).

Любая программа на Python может считаться модулем (все те программы, которые мы писали, можно назвать модулями). Это отличает Python от Pascal и его наследников, где модуль и главная программа оформляются немного различно. В этой теме упорядочим все способы подключения стандартных модулей для Python, а также научимся создавать свои модули и подключать их. Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам. Нужно заметить, что модуль может быть написан не только на Python, а например, на С или C++.

Первый способ подключить модуль: с помощью инструкции `import`.

После ключевого слова `import` указывается название модуля. Одной инструкцией можно подключить несколько модулей, хотя этого не рекомендуется делать, так как это снижает читаемость кода:

```
import math
import numpy, scipy
```

После импортирования модуля его название становится переменной, через которую можно получить доступ к атрибутам модуля. Например, можно обратиться к константе `e`, расположенной в модуле `math`:

```
1 import math
2 print(math.e)
3 print(math.pi)
```

→ 2.718281828459045  
3.141592653589793

Стоит отметить, что если указанный атрибут модуля не будет найден, возбудится исключение `AttributeError`. А если не удастся найти модуль для импортирования, то `ImportError`:

```
1 import foo
```

→ -----  
ModuleNotFoundError Traceback (most recent call last)  
<ipython-input-6-7f58dd7fb72e> in <cell line: 1>()  
----> 1 import foo  
  
ModuleNotFoundError: No module named 'foo'

Второй способ подключить модуль: использовать псевдонимы.

Если название модуля слишком длинное, или оно вам не нравится по каким-то другим причинам, то для него можно создать псевдоним, с помощью ключевого слова `as`.

```
1 import math as m
2 # sin(30)
3 print(m.sin(30 * m.pi / 180))
```

→ 0.4999999999999994

Третий способ подключить модуль: с помощью инструкции `from` (первый формат). Подключить определенные атрибуты модуля можно с помощью инструкции `from`. Она имеет несколько форматов:

```
from <Название модуля> import <Атрибут 1> as <Псевдоним 1>,
<Атрибут 2> as <Псевдоним 2>, ...
```

Пример:

```
▶ 1 from math import ceil as c, pi as PI, factorial as F
 2 print(c(34.8))
 3 print(PI)
 4 print(F(5))

▶ 35
3.141592653589793
120
```

Четвёртый способ подключить модуль: с помощью инструкции `from` (второй формат).

```
from <Название модуля> import *
```

Данный формат инструкции `from` позволяет подключить все (точнее, почти все) переменные из модуля.

## 9.2. Стандартная библиотека Python. Установка библиотек

Стандартная библиотека Python очень обширна и предлагает широкий спектр возможностей. Библиотека содержит встроенные модули (написанные на языке C), обеспечивающие доступ к таким функциональным возможностям системы, как файловый ввод-вывод, а также модули, написанные на языке Python, обеспечивающие стандартные решения многих проблем, возникающих при повседневном программировании. Некоторые из этих модулей специально разработаны для поощрения и повышения переносимости Python программ путем абстракции особенностей платформы в виде нейтрального к платформе API.

Установщики Python для платформы Windows обычно включают всю стандартную библиотеку и часто также включают множество дополнительных компонентов. Для Unix-подобных операционных систем Python обычно

предоставляется в виде набора пакетов, поэтому может потребоваться использование пакетных инструментов, поставляемые с операционной системой, для получения некоторых или всех необязательных компонентов.

Помимо стандартной библиотеки, существует растущая коллекция из нескольких тысяч компонентов (от отдельных программ и модулей до пакетов и фреймворков разработки приложений).

У Python интерпретатора есть ряд встроенных в него функций и типов, которые доступны всегда (таблица 9.1).

Таблица 9.1 – Встроенные функции Python

|                            |                          |                           |                           |                             |
|----------------------------|--------------------------|---------------------------|---------------------------|-----------------------------|
| <code>abs()</code>         | <code>delattr()</code>   | <code>hash()</code>       | <code>memoryview()</code> | <code>set()</code>          |
| <code>all()</code>         | <code>dict()</code>      | <code>help()</code>       | <code>min()</code>        | <code>setattr()</code>      |
| <code>any()</code>         | <code>dir()</code>       | <code>hex()</code>        | <code>next()</code>       | <code>slice()</code>        |
| <code>ascii()</code>       | <code>divmod()</code>    | <code>id()</code>         | <code>object()</code>     | <code>sorted()</code>       |
| <code>bin()</code>         | <code>enumerate()</code> | <code>input()</code>      | <code>oct()</code>        | <code>staticmethod()</code> |
| <code>bool()</code>        | <code>eval()</code>      | <code>int()</code>        | <code>open()</code>       | <code>str()</code>          |
| <code>breakpoint()</code>  | <code>exec()</code>      | <code>isinstance()</code> | <code>ord()</code>        | <code>sum()</code>          |
| <code>bytearray()</code>   | <code>filter()</code>    | <code>issubclass()</code> | <code>pow()</code>        | <code>super()</code>        |
| <code>bytes()</code>       | <code>float()</code>     | <code>iter()</code>       | <code>print()</code>      | <code>tuple()</code>        |
| <code>callable()</code>    | <code>format()</code>    | <code>len()</code>        | <code>property()</code>   | <code>type()</code>         |
| <code>chr()</code>         | <code>frozenset()</code> | <code>list()</code>       | <code>range()</code>      | <code>vars()</code>         |
| <code>classmethod()</code> | <code>getattr()</code>   | <code>locals()</code>     | <code>repr()</code>       | <code>zip()</code>          |
| <code>compile()</code>     | <code>globals()</code>   | <code>map()</code>        | <code>reversed()</code>   | <code>__import__()</code>   |
| <code>complex()</code>     | <code>hasattr()</code>   | <code>max()</code>        | <code>round()</code>      |                             |

Описание каждой функции можно найти в многочисленной онлайн-документации или воспользоваться функцией `help`. Получение справки о функции `bin` с использованием встроенной функции `help`:

```

 1 print(help(bin))

Help on built-in function bin in module builtins:

bin(number, /)
 Return the binary representation of an integer.

>>> bin(2796202)
'0b10101010101010101010101010'

```

Модули библиотеки Python представлены в таблице 9.2.

Таблица 9.2 – Модули стандартной библиотеки

| Модуль          | Описание                                                     |
|-----------------|--------------------------------------------------------------|
| datetime        | Базовые типы для представления даты и времени                |
| calendar        | Календарные функции                                          |
| collections     | Контейнерные типы данных                                     |
| collections.abc | Абстрактные базовые классы для контейнеров                   |
| heapq           | Алгоритм очереди кучи                                        |
| bisect          | Алгоритм деления массива пополам                             |
| array           | Эффективные массивы числовых значений                        |
| weakref         | Слабые ссылки                                                |
| types           | Создание динамического типа и имени для встроенных типов     |
| copy            | Функции поверхностного и глубокого копирования               |
| pprint          | Приятная печать данных                                       |
| reprlib         | Альтернативная реализация repr()                             |
| enum            | Поддержка перечислений                                       |
| numbers         | Числовые абстрактные базовые классы                          |
| math            | Математические функции                                       |
| cmath           | Математические функции для комплексных чисел                 |
| decimal         | Десятичная арифметика с фиксированной и плавающей точкой     |
| fractions       | Рациональные числа                                           |
| random          | Генерация псевдослучайных чисел                              |
| statistics      | Функции математической статистики                            |
| itertools       | Функции, создающие итераторы для эффективного цикла          |
| functools       | Функции и операции высшего порядка над вызываемыми объектами |
| operator        | Стандартные операторы как функции                            |
| pathlib         | Объектно-ориентированные пути файловой системы               |
| os.path         | Общие манипуляции с путями к файлам и каталогам              |
| fileinput       | Перебор строк из нескольких входных потоков                  |
| stat            | Интерпретация результатов stat()                             |
| filecmp         | Сравнение файлов и каталогов                                 |
| tempfile        | Создание временных файлов и каталогов                        |
| glob            | Расширение шаблона имени пути в стиле Unix                   |
| fnmatch         | Соответствие шаблону имени файла Unix                        |
| linecache       | Произвольный доступ к строкам текста                         |

|              |                                                              |
|--------------|--------------------------------------------------------------|
| shutil       | Высокоуровневые файловые операции                            |
| pickle       | Сериализация Python объекта                                  |
| copyreg      | Регистрация поддерживающей pickle функции                    |
| shelve       | Сохраняемость Python объектов                                |
| marshal      | Внутренняя сериализация Python объектов                      |
| dbm          | Интерфейсы к «базам данных» Unix                             |
| sqlite3      | Интерфейс DB-API 2.0 для баз данных SQLite                   |
| zlib         | Сжатие совместимое с gzip                                    |
| gzip         | Поддержка gzip файлов                                        |
| bz2          | Поддержка сжатия bzip2                                       |
| lzma         | Сжатие LZMA алгоритмом                                       |
| zipfile      | Работа с ZIP-архивами                                        |
| tarfile      | Чтение и запись tar архивных файлов                          |
| csv          | Чтение и запись CSV файлов                                   |
| configparser | Парсер конфигурационного INI файла                           |
| netrc        | Обработка netrc файла                                        |
| xdrlib       | Кодирование и декодирование XDR данных                       |
| plistlib     | Создание и парсинг файлов Mac OS .plist                      |
| hashlib      | Безопасные хэши и дайджесты сообщений                        |
| hmac         | Хэширование с ключом для аутентификации сообщений            |
| secrets      | Создание безопасных случайных чисел для управления секретами |
| tkinter      | Python интерфейс для Tcl/Tk                                  |
| webbrowser   | Удобный контроллер веб-браузера                              |
| http         | HTTP модули                                                  |
| html         | Поддержка языка гипертекстовой разметки                      |
| email        | Пакет обработки электронной почты и MIME                     |
| json         | Кодер и декодер JSON                                         |

В таблице 9.2 показаны только некоторые модули. Полный перечень (более 200) можно найти в онлайн-документации Python.

### 9.3. Модуль os

Рассмотрим пример использования модулей, которые наиболее часто применяются при разработке на Python.

Модуль **os** обеспечивает переносимый способ использования функций, зависящих от операционной системы.

Перечень инструментов, которыми обеспечивает **os** программистов:

**os.name** – имя операционной системы. Доступные варианты: '**posix**', '**nt**', '**mac**', '**os2**', '**ce**', '**java**'.

**os.environ** – словарь переменных окружения. Изменяемый (можно добавлять и удалять переменные окружения).

**os.getpid()** – текущий id процесса.

**os.uname()** – информация об ОС. возвращает объект с атрибутами: **sysname** – имя операционной системы, **nodename** – имя машины в сети (определяется реализацией), **release** – релиз, **version** – версия, **machine** – идентификатор машины.

**os.chdir(path)** – смена текущей директории.

**os.getcwd()** – текущая рабочая директория.

**os.listdir(path=".")** – список файлов и директорий в папке.

**os.mkdir(path, mode=0o777, \*, dir\_fd=None)** – создаёт директорию. **OSError**, если директория существует.

**os.makedirs(path, mode=0o777, exist\_ok=False)** – создаёт директорию, создавая при этом промежуточные директории.

**os.remove(path, \*, dir\_fd=None)** – удаляет путь к файлу.

**os.rename(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None)** – переименовывает файл или директорию из **src** в **dst**.

**os.replace(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None)** – переименовывает из **src** в **dst** с принудительной заменой.

**os.rmdir(path, \*, dir\_fd=None)** – удаляет пустую директорию.

**os.removedirs(path)** – удаляет директорию, затем пытается удалить родительские директории, и удаляет их рекурсивно, пока они пусты.

**os.system(command)** – исполняет системную команду, возвращает код её завершения (в случае успеха 0).

**os.path** – модуль, реализующий некоторые полезные функции на работы с путями.

Пример кода с использованием возможностей модуля **os**:

```
1 import os
2 print(os.name)
3 print(os.getcwd())
4 print(os.getpid())
```

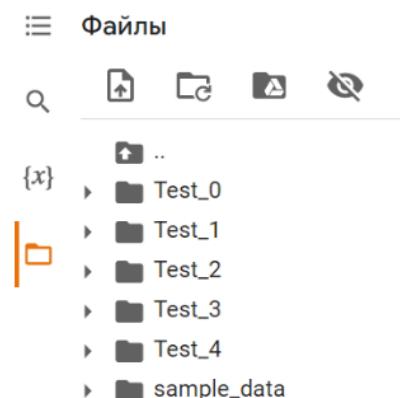
posix  
/content  
238

Следующий пример демонстрирует создание каталогов и файлов:

```
1 import os, os.path
2
3 current_path = os.getcwd()
4 dir_name = 'Test'
5
6 for i in range(5):
7 p = os.path.join(current_path, f'{dir_name}_{i}')
8 if not os.path.exists(p):
9 os.mkdir(p)
10 print(f'Каталог создан: {p}')
```

Каталог создан: /content/Test\_0  
 Каталог создан: /content/Test\_1  
 Каталог создан: /content/Test\_2  
 Каталог создан: /content/Test\_3  
 Каталог создан: /content/Test\_4

В результате выполнения данного скрипта, в рабочем каталоге на виртуальной машине будет создана следующая структура:



## 9.4. Модуль math

Возможности модуля **math** уже использовались в данном курсе. Модуль содержит математические функции:

**math.ceil(X)** – округление до ближайшего большего числа.

**math.copysign(X, Y)** – возвращает число, имеющее модуль такой же, как и у числа X, а знак – как у числа Y.

**math.fabs(X)** – модуль X.

**math.factorial(X)** – факториал числа X.

**math.floor(X)** – округление вниз.

**math.fmod(X, Y)** – остаток от деления X на Y.

**math.frexp(X)** – возвращает мантиссу и экспоненту числа.

**math.ldexp(X, I)** –  $X * 2^I$ . Функция, обратная функции **math.frexp()**.

**math.fsum(последовательность)** – сумма всех членов последовательности. Эквивалент встроенной функции **sum()**, но **math.fsum()** более точна для чисел с плавающей точкой.

**math.isfinite(X)** – является ли X числом.

**math.isinf(X)** – является ли X бесконечностью.

**math.isnan(X)** – является ли X **NaN** (Not a Number – не число).

**math.modf(X)** – возвращает дробную и целую часть числа X. Оба числа имеют тот же знак, что и X.

**math.trunc(X)** – усекает значение X до целого.

$\text{math.exp}(X) = e^X$ .

**math.expm1(X)** –  $e^X - 1$ . При  $X \rightarrow 0$  точнее, чем **math.exp(X)-1**.

**math.log(X, [base])** – логарифм X по основанию base. Если base не указан, вычисляется натуральный логарифм.

**math.log1p(X)** – натуральный логарифм  $(1 + X)$ . При  $X \rightarrow 0$  точнее, чем **math.log(1+X)**.

**math.log10(X)** – логарифм X по основанию 10.

**math.log2(X)** – логарифм X по основанию 2.

**math.pow(X, Y)** –  $X^Y$ .

**math.sqrt(X)** – квадратный корень из X.

**math.acos(X)** – арккосинус X. В радианах.

**math.asin(X)** – арксинус X. В радианах.

`math.atan(X)` – арктангенс X. В радианах.

`math.atan2(Y, X)` – арктангенс Y/X. В радианах. С учетом четверти, в которой находится точка (X, Y).

`math.cos(X)` – косинус X (X указывается в радианах).

`math.sin(X)` – синус X (X указывается в радианах).

`math.tan(X)` – тангенс X (X указывается в радианах).

`math.hypot(X, Y)` – вычисляет гипотенузу треугольника с катетами X и Y (`math.sqrt(x * x + y * y)`).

`math.degrees(X)` – конвертирует радианы в градусы.

`math.radians(X)` – конвертирует градусы в радианы.

`math.cosh(X)` – вычисляет гиперболический косинус.

`math.sinh(X)` – вычисляет гиперболический синус.

`math.tanh(X)` – вычисляет гиперболический тангенс.

`math.acosh(X)` – вычисляет обратный гиперболический косинус.

`math.asinh(X)` – вычисляет обратный гиперболический синус.

`math.atanh(X)` – вычисляет обратный гиперболический тангенс.

`math.erf(X)` – функция ошибок.

`math.erfc(X)` – дополнительная функция ошибок (`1 - math.erf(X)`).

`math.gamma(X)` – гамма-функция X.

`math.lgamma(X)` – натуральный логарифм гамма-функции X.

`math.pi` –  $\pi = 3,1415926\dots$

`math.e` –  $e = 2,718281\dots$

Пример вычислений с использованием модуля `math` (вывод таблицы синусов и косинусов):

```
1 from math import sin, cos, tan, radians
2 print('\t0 \t30 \t60 \t90')
3
4 print('sin', end='\t')
5 for angle in range(0, 100, 30):
6 print(f'{sin(radians(angle)) :.2f} ', end='\t')
7 print()
```

```

8 print('cos', end='\t')
9 for angle in range(0, 100, 30):
10 print(f'{cos(radians(angle)) :.2f} ', end='\t')
11 print()
12 print('tg', end='\t')
13 for angle in range(0, 100, 30):
14 print(f'{tan(radians(angle)) :.2f} ', end='\t')

→ 0 30 60 90
sin 0.00 0.50 0.87 1.00
cos 1.00 0.87 0.50 0.00
tg 0.00 0.58 1.73 16331239353195370.00

```

## 9.5. Модуль `itertools`

Данное расширение является сборником полезных итераторов, повышающих эффективность работы с циклами и генераторами последовательностей объектов. Это достигается за счет лучшего управления памятью в программе, быстрого выполнения подключаемых функций, а также сокращения и упрощения кода. Готовые методы, реализованные в данной библиотеке, принимают различные параметры для управления генератором последовательности, чтобы вернуть вызывающей подпрограмме необходимый набор объектов.

Основные возможности `itertools`:

`count(start=0, step=1)` – бесконечная арифметическая прогрессия с первым членом `start` и шагом `step`.

`cycle(iterable)` – возвращает по одному значению из последовательности, повторенной бесконечное число раз.

`repeat(elem, n=Inf)` – повторяет `elem` `n` раз.

`accumulate(iterable)` – аккумулирует суммы.

`chain(*iterables)` – возвращает по одному элементу из первого итератора, потом из второго, до тех пор, пока итераторы не кончатся.

`combinations(iterable, [r])` – комбинации длиной `r` из `iterable` без повторяющихся элементов.

**combinations\_with\_replacement(iterable, r)** – комбинации длиной r из iterable с повторяющимися элементами.

**compress(data, selectors)** – (d[0] if s[0]), (d[1] if s[1]), ...

**dropwhile(func, iterable)** – элементы iterable, начиная с первого, для которого func вернула ложь.

**filterfalse(func, iterable)** – все элементы, для которых func возвращает ложь.

**groupby(iterable, key=None)** – группирует элементы по значению. Значение получается применением функции key к элементу (если аргумент key не указан, то значением является сам элемент).

**islice(iterable[, start], stop[, step])** – итератор, состоящий из среза.

**permutations(iterable, r=None)** – перестановки длиной r из iterable.

**product(\*iterables, repeat=1)** – аналог вложенных циклов.

**starmap(function, iterable)** – применяет функцию к каждому элементу последовательности (каждый элемент распаковывается).

**takewhile(func, iterable)** – элементы до тех пор, пока func возвращает истину.

**zip\_longest(\*iterables, fillvalue=None)** – какстроенная функция zip, но берет самый длинный итератор, а более короткие дополняет fillvalue.

Рассмотрим пример работы с библиотекой itertools. В следующем примере реализуется бесконечный цикл с использованием **cycle**, который обходит последовательность циклически:

```
1 from itertools import cycle
2 names = ['Евгений', 'Мария', 'Алекс', 'Макс', 'Оля']
3
4 for name in cycle(names):
5 print(name)
6 answer = input('Достаточно ? (Д/Н) > ')
7 if answer.upper() == 'Д':
8 break
```

```
→ Евгений
Достаточно ? (Д/Н) > н
Мария
Достаточно ? (Д/Н) > н
Алекс
Достаточно ? (Д/Н) > н
Макс
Достаточно ? (Д/Н) > н
Оля
Достаточно ? (Д/Н) > н
Евгений
Достаточно ? (Д/Н) > н
Мария
Достаточно ? (Д/Н) > д
```

С использованием модуля `itertools` легко решать комбинаторные задачи.

Например, необходимо посчитать количество различных четырехбуквенных слов, которые можно составить из букв С, Л, О, Н, И, Х, А, причем каждую букву можно использовать любое количество раз и получаемые слова не обязательно должны быть осмыслившими словами русского языка.

Например, подходят слова:

CCCC, СССЛ, ССОН, СЛОН, НОХХ и т.д.

Данная задача легко решается с использованием функции `product` библиотеки `itertools`. Напишем код, который выводит все возможные слова:

```
▶ 1 from itertools import product
 2 alphabet = 'СЛОНИХА'
 3 for word in product(alphabet, repeat=4):
 4 print(word)
```

```
→ ('С', 'С', 'С', 'С')
('С', 'С', 'С', 'Л')
('С', 'С', 'С', 'О')
('С', 'С', 'С', 'Н')
('С', 'С', 'С', 'И')
('С', 'С', 'С', 'Х')
('С', 'С', 'С', 'А')
('С', 'С', 'Л', 'С')
('С', 'С', 'Л', 'Л')
('С', 'С', 'Л', 'О')
```

После просмотра слов, можно заняться их подсчетом (выводить в принципе не обязательно):



```

1 from itertools import product
2 alphabet = 'СЛОНИХА'
3 counter=0
4 for word in product(alphabet, repeat=4):
5 counter+=1
6 print(counter, 7**4)

```

⇨ 2401 2401

Обратите внимание, что ответ для данной задачи  $7^4$  может быть получен аналитически. Можно решить данную задачу без явного использования циклов:



```

1 from itertools import product
2 alphabet = 'СЛОНИХА'
3 print(len(list(product(alphabet, repeat=4))))

```

⇨ 2401

## 9.6. Модуль random

Модуль `random` предоставляет функции для генерации случайных чисел, букв, случайного выбора элементов последовательности. Основные возможности:

`seed([X], version=2)` – инициализация генератора случайных чисел.

Если X не указан, используется системное время.

`getstate()` – внутреннее состояние генератора.

`setstate(state)` – восстанавливает внутреннее состояние генератора.

Параметр `state` должен быть получен функцией `getstate()`.

`getrandbits(N)` – возвращает N случайных бит.

`randrange(start, stop, step)` – возвращает случайно выбранное число из последовательности.

`randint(A, B)` – случайное целое число N, A ≤ N ≤ B.

`choice(sequence)` – случайный элемент непустой последовательности.

**shuffle(sequence, [rand])** – перемешивает последовательность (изменяется сама последовательность). Поэтому функция не работает для неизменяемых объектов.

**sample(population, k)** – список длиной **k** из последовательности **population**.

**random()** – случайное число от 0 до 1.

**uniform(A, B)** – случайное число с плавающей точкой, **A ≤ N ≤ B** (или **B ≤ N ≤ A**).

**triangular(low, high, mode)** – случайное число с плавающей точкой, **low ≤ N ≤ high**. **Mode** – распределение.

**betavariate(alpha, beta)** – бета-распределение. **alpha>0, beta>0**.

Возвращает от 0 до 1.

**expovariate(lambd)** – экспоненциальное распределение. **lambd** равен **1/среднее желаемое**; **lambd** должен быть отличным от нуля. Возвращаемые значения от 0 до плюс бесконечности, если **lambd** положительно, и от минус бесконечности до 0, если **lambd** отрицательный.

**gammavariate(alpha, beta)** – гамма-распределение. Условия на параметры **alpha>0** и **beta>0**.

**gauss(значение, стандартное отклонение)** – распределение Гаусса.

**lognormvariate(mu, sigma)** – логарифм нормального распределения. Если взять натуральный логарифм этого распределения, то вы получите нормальное распределение со средним **mu** и стандартным отклонением **sigma**. **mu** может иметь любое значение, и **sigma** должна быть больше нуля.

**normalvariate(mu, sigma)** – нормальное распределение; **mu** – среднее значение, **sigma** – стандартное отклонение.

**vonmisesvariate(mu, kappa)** – где **mu** – средний угол, выраженный в радианах от 0 до  $2\pi$ , и **kappa** – параметр концентрации, который должен быть больше или равен нулю. Если каппа равна нулю, это распределение сводится к случайному углу в диапазоне от 0 до  $2\pi$ .

**paretovariate(alpha)** – распределение Парето.

`weibullvariate(alpha, beta)` – распределение Вейбулла.

Чаще всего в программировании требуется получать наборы случайных целых и вещественных чисел. Реализуем программу, которая создает текстовый файл и заполняет его строками псевдослучайных данных о сотрудниках в формате: ФИО – возраст – зарплата.

```
1 from random import randint, random, choice, seed
2 names=['Иванов', 'Петров', 'Сидоров', \
3 'Поляков', 'Войтик', 'Серов',
4 'Пунькин', 'Солеев', 'Кирина']
5 seed(5)
6 file_output = open('data.txt', 'w')
7 N = 15
8 for _ in range(N):
9 name = choice(names)
10 age = randint(18, 80)
11 salary = round(random()*10000, 2)
12 file_output.write(f'{name} {age} {salary}\n')
13 file_output.close()
```

В результате выполнения скрипта получится файл, содержащий следующие данные:

| data.txt |         |            |
|----------|---------|------------|
| 1        | Войтик  | 65 3585.36 |
| 2        | Кирина  | 19 8403.48 |
| 3        | Поляков | 59 518.53  |
| 4        | Сидоров | 25 3717.93 |
| 5        | Поляков | 42 5437.61 |
| 6        | Поляков | 18 7311.84 |
| 7        | Пунькин | 35 1820.76 |
| 8        | Пунькин | 28 7619.25 |
| 9        | Петров  | 26 6179.26 |
| 10       | Солеев  | 26 1322.41 |
| 11       | Иванов  | 31 7735.94 |
| 12       | Сидоров | 73 1664.84 |
| 13       | Серов   | 79 1988.89 |
| 14       | Поляков | 29 9409.76 |
| 15       | Поляков | 79 9650.96 |
| 16       |         |            |

## 9.7. Пользовательские модули

Программист может самостоятельно создавать модули и подключать их по необходимости. Это делает программу более читабельной, структуру программы более понятной, такие программы легче сопровождать и модифицировать.

Рассмотрим задачу:

Требуется написать приложение, которое по данным, содержащимся в исходном файле вычисляет суммарную налогооблагаемую базу (сумму с которой производится начисление налога), а также сумму начисленного налога. Фрагмент файла представлен на рис. 9.1.

|    | ID, FName, LName, Company, Salary, Language                    |
|----|----------------------------------------------------------------|
| 1  | 67-9921732, Giorgi, Kirvell, Zuko Ltd, \$1018.20, Italian      |
| 2  | 04-8410537, Page, Leere, Workus Co, \$1716.63, English         |
| 3  | 73-6146335, Corrie, Beste, Workus Co, \$4738.98, English       |
| 4  | 59-8017461, Derril, Raitie, Workus Co, \$1873.63, Italian      |
| 5  | 19-3865975, Walsh, Linster, Craft Ltd, \$1618.47, English      |
| 6  | 55-0884283, Brion, Mosconi, NodaTLT Co, \$1066.75, Russian     |
| 7  | 11-4873482, Jude, Beneix, Workus Co, \$2677.57, Russian        |
| 8  | 64-1959092, Karlis, Talks, NodaTLT Co, \$1843.45, French       |
| 9  | 37-9347477, Jay, Kerton, NodaTLT Co, \$1950.97, Russian        |
| 10 | 65-4588299, Prince, Sexton, Zuko Ltd, \$2268.92, Italian       |
| 11 | 89-7195753, Delmer, D'Agostini, NodaTLT Co, \$2230.91, Russian |
| 12 | 07-3467105, Raffarty, Hodge, Zuko Ltd, \$2939.58, English      |
| 13 | 34-9024489, Syd, LLelweln, Zuko Ltd, \$1074.51, English        |
| 14 | 75-7595742, Orv, Martt, Zuko Ltd, \$1346.83, Russian           |
| 15 | 88-3861710, Colby, Nash, Zuko Ltd, \$4361.76, French           |
| 16 | 87-0025080, Tyrone, Bradd, NodaTLT Co, \$2835.00, English      |
| 17 | 90-7599554, Elden, Trapp, NodaTLT Co, \$3255.45, Russian       |
| 18 |                                                                |

Рисунок 9.1 – Фрагмент входного файла

В представленном файле 1000 строк. Так как в рамках данного курса используется Google Colab, то необходимо поместить входной файл на сервер, где он будет доступен для чтения. Файл размещен в репозитории github, поэтому с использованием команды wget происходит скачивание текстового файла:

```

1 # магические команды Python
2 # скачиваем файл с данными по URL
3 !wget https://raw.githubusercontent.com/enikolaev/Algorithmization_and_programming/main/data/data_theme9.csv

--2023-09-03 09:41:53-- https://raw.githubusercontent.com/enikolaev/Algorithmization_and_programming/main/data/data_theme9.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133, 185.199.108.133, 185.199.109.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 53585 (52K) [text/plain]
Saving to: 'data_theme9.csv'

data_theme9.csv 100%[=====] 52.33K --.-KB/s in 0.01s

2023-09-03 09:41:53 (4.32 MB/s) - 'data_theme9.csv' saved [53585/53585]

```

После этого можно решать задачу... Создадим отдельный модуль для разбора строки из файла и отдельный модуль для финансовых расчетов (вычисление налога). В рабочем каталоге сервера создадим файл **finance.py** и **parser.py**:

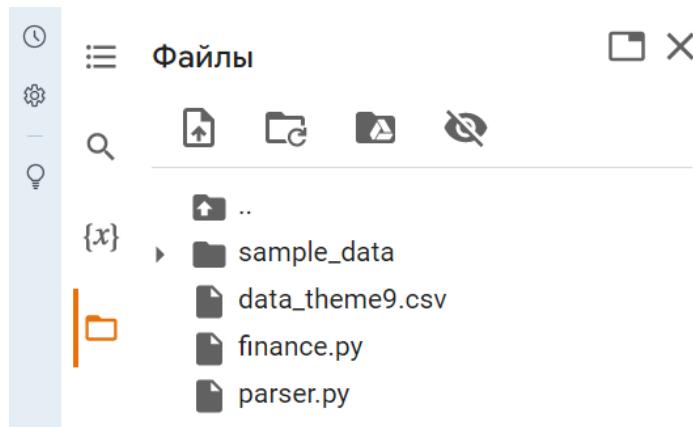


Рисунок 9.2 – Отдельные файлы на виртуальном сервере

В файл **finance.py** вынесем следующий код:

```

finance.py X ...
1 Stavka = 13
2
3 def get_nalog(salary):
4 return round(salary * Stavka / 100, 2)

```

В файл **parser.py** необходимо добавить следующий код:

```

parser.py X ...
1 def parse(line):
2 p=line.split(',')
3 d={'id':p[0], 'fio': p[1]+' '+p[2] +' '+p[3], \
4 'salary':float(p[4][1:]), 'lang':p[5]}
5 return d

```

После определения модулей, воспользуемся ими как стандартными библиотеками:

```

1 import finance as F
2 import parser as P
3
4 f = open('data_theme9.csv')
5 sum_nalog = 0
6 f.readline()
7 for line in f:
8 data = P.parse(line)
9 sum_nalog += data['salary']
10 f.close()
11
12 print(f'Налогооблагаемая база: {round(sum_nalog, 2)}')
13 print(f'Налогооблагаемая база: {round(F.get_nalog(sum_nalog), 2)}')

```

⇨ Налогооблагаемая база: 3013560.18  
Налогооблагаемая база: 391762.82

Данный пример слишком простой для разработки многомодульного приложения, но общая концепция ясна – разделение приложения на отдельные модули для организации нисходящего проектирования. То есть решаем отдельные частные подзадачи большой сложной задачи.

### **Вопросы для самопроверки по теме 9**

1. Что такое модуль в Python?
2. Сколько модулей может подключить программист?
3. С использованием какой команды производится подключение модуля?
4. Перечислите основные библиотеки Python и опишите их назначение.

## Лекция 10. Объектно-ориентированное программирование

### План лекции

1. Процедурное и объектно-ориентированное.
2. Класс.
3. Объект. Экземпляры класса.
4. Консервация пользовательских объектов

### 10.1. Процедурное и объектно-ориентированное

Процедурное программирование представляет собой метод написания программного обеспечения. Это практика программирования, в центре внимания которой находятся процедуры или действия, происходящие в программе. Основой объектно-ориентированного программирования служат объекты, которые создаются из абстрактных типов данных, объединяющих данные и функции.

В настоящее время применяются главным образом два метода программирования: процедурный и объектно-ориентированный. Первые языки программирования были процедурными, т. е. программа состояла из одной или нескольких процедур. Процедура может рассматриваться просто как функция, которая выполняет определенную задачу, такую как сбор вводимых пользователем данных, выполнение вычислений, чтение или запись файлов, вывод результатов и т. д. Программы, которые вы писали до сих пор, были по своей природе процедурными.

Как правило, процедуры оперируют элементами данных, которые существуют отдельно от процедур. В процедурной программе элементы данных обычно передаются из одной процедуры в другую. Как можно предположить, в центре процедурного программирования находится создание процедур, которые оперируют данными программы. По мере увеличения и усложнения программы разделение данных и программного кода, который оперирует данными, может привести к проблемам.

Например, предположим, что вы являетесь участником команды программистов, которая написала масштабную программу обработки базы данных клиентов. Эта программа первоначально разрабатывалась с использованием трех переменных, которые ссылаются на имя, адрес и телефонный номер клиента. Ваша работа состояла в том, чтобы разработать несколько функций, которые принимают эти три переменные в качестве аргументов и выполняют с ними операции. Созданный программный продукт успешно работал в течение некоторого времени, однако вашу команду попросили его обновить, внедрив несколько новых возможностей. Во время пересмотра версии ведущий программист вам сообщает, что имя, адрес и телефонный номер клиента больше не будут храниться в переменных. Вместо этого они будут храниться в списке. Это означает, что вам необходимо изменить все разработанные вами функции таким образом, чтобы они принимали список и работали с ним вместо этих трех переменных. Внесение таких масштабных модификаций не только предполагает большой объем работы, но и открывает возможность для внесения ошибок в программный код.

В отличие от процедурного программирования, в центре внимания которого находится создание процедур (функций), объектно-ориентированное программирование (ООП) сосредоточено на создании объектов. Объект – это программная сущность, которая содержит данные и процедуры. Находящиеся внутри объекта данные называются атрибутами данных.

Это просто переменные, которые ссылаются на данные. Выполняемые объектом процедуры называются методами. Методы объекта – это функции, которые выполняют операции с атрибутами данных. В концептуальном плане объект представляет собой автономную единицу, которая состоит из атрибутов данных и методов, которые оперируют атрибутами данных (рис. 10.1).

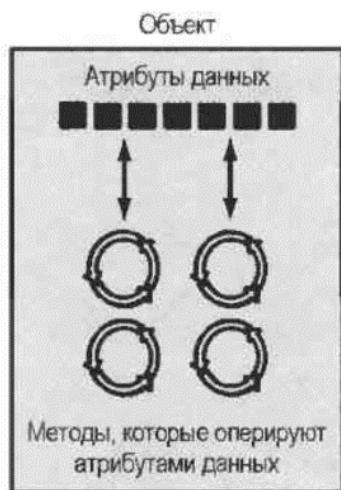


Рисунок 10.1 – Объект с атрибутами данных и методами

ООП решает проблему разделения программного кода и данных посредством инкапсуляции и скрытия данных. Инкапсуляция обозначает объединение данных и программного кода в одном объекте. Скрытие данных связано со способностью объекта скрывать свои атрибуты данных от программного кода, который находится за пределами объекта. Только методы объекта могут непосредственно получать доступ и вносить изменения в атрибуты данных объекта.

Объект, как правило, скрывает свои данные, но позволяет внешнему коду получать доступ к своим методам. Как показано на рис. 10.2, методы объекта предоставляют программным инструкциям за пределами объекта косвенный доступ к атрибутам данных объекта.

Когда атрибуты данных объекта скрыты от внешнего кода, и доступ к атрибутам данных ограничен методами объекта, атрибуты данных защищены от случайного повреждения.

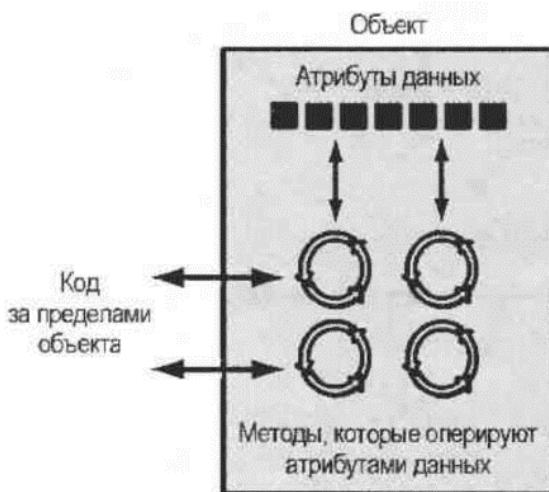


Рисунок 10.2 – Программный код за пределами объекта взаимодействует с методами объекта

Кроме того, программному коду за пределами объекта не нужно знать о формате или внутренней структуре данных объекта. Программный код взаимодействует только с методами объекта. Когда программист меняет структуру внутренних атрибутов данных, он также меняет методы объекта, чтобы они могли должным образом оперировать данными. Однако приемы взаимодействия внешнего кода с методами не меняются.

#### 10.1.1. Возможность многократного использования объекта

В дополнение к решению проблем разделения программного кода и данных, применение ООП также всецело поддерживалось трендом на многократное использование объектов. Объект не является автономной программой. Напротив, он используется программами, которым нужны его услуги. Например, **Программист1** является программистом, и он разработал ряд объектов для визуализации 3D-изображений. Он эрудит в математике и очень много знает о компьютерной графике, поэтому его объекты запрограммированы на выполнение всех необходимых математических операций с 3D-графикой и взаимодействие с компьютерным видеооборудованием. Для **Программист2**, который пишет программу по заказу архитектурной фирмы, требуется, чтобы его приложение выводило 3D-

изображения зданий. Поскольку он работает в рамках жестких сроков и не обладает большим объемом знаний в области компьютерной графики, то может применить объекты **Программист1**, чтобы выполнить 3D-визуализацию (за небольшую плату, разумеется!).

### 10.1.2. Пример объекта из повседневной жизни

Предположим, что будильник – это на самом деле программный объект. Будь это так, то он имел бы приведенные ниже атрибуты данных:

- ◆ **current\_second** (текущая секунда, значение в диапазоне 0-59);
- ◆ **current\_minute** (текущая минута, значение в диапазоне 0-59);
- ◆ **current\_hour** (текущий час, значение в диапазоне 1-12);
- ◆ **alarm\_time** (время сигнала, допустимые час и минута);
- ◆ **alarm\_is\_set** (будильник включен, истина или ложь).

Этот пример четко показывает, что атрибуты данных – это всего-навсего значения, которые определяют состояние, в котором будильник находится в настоящее время. Вы, пользователь объекта «Будильник», не можете непосредственно манипулировать этими атрибутами данных, потому что они являются приватными, или частными. Для того чтобы изменить значение атрибута данных, необходимо применить один из методов объекта. Ниже приведено несколько методов объекта «Будильник»:

- ◆ **set\_time( )** (задать время часов);
- ◆ **set\_alarm\_time( )** (задать время сигнала);
- ◆ **set\_alarm\_on( )** (включить будильник);
- ◆ **set\_alarm\_off( )** (выключить будильник).

Каждый метод манипулирует одним или несколькими атрибутами данных. Например, метод **set\_time( )** позволяет устанавливать время будильника и активируется нажатием кнопки вверху часов. При помощи другой кнопки можно активировать метод **set\_alarm\_time( )**.

Еще одна кнопка позволяет выполнять методы **set\_alarm\_on( )** и **set\_alarm\_off( )**. Обратите внимание, что все эти методы могут быть

активированы вами, т. е. тем, кто находится за пределами будильника. Методы, к которым могут получать доступ объекты, находящиеся за пределами объекта, называются открытыми, или публичными, методами.

Будильник также имеет закрытые, или приватные, методы, которые являются составной частью приватного, внутреннего устройства объекта. Внешние сущности (такие как вы, пользователь будильника) не имеют прямого доступа к приватным методам будильника.

Объект предназначен выполнять эти методы автоматически и скрывать детали от вас. Ниже приведены приватные методы объекта «Будильник» (сугубо для внутреннего использования):

- ◆ `imeregent_current_second( )` (прирастить текущую секунду);
- ◆ `imcrement_current_minute( )` (прирастить текущую минуту);
- ◆ `increment_current_hour( )` (прирастить текущий час);
- ◆ `sound_alarm( )` (подать звуковой сигнал).

Метод `imeregent_current_second( )` выполняется каждую секунду. Он изменяет значение атрибута данных `current_second`. Если при исполнении этого метода атрибут данных `current second` равняется **59**, то этот метод запрограммирован сбросить `current_second` в значение **0** и затем вызвать метод `increment_current_minute( )`, который добавляет 1 к атрибуту данных `current_minute`, если он не равен **59**. В противном случае он сбрасывает `current_minute` в значение **0** и вызывает метод `increment_current_hour( )`. Метод `increment_current_minute( )` сравнивает новое время с `alarm_time`. Если оба времени совпадают и при этом звонок включен, исполняется метод `sound_alarm( )`.

## 10.2. Класс

Класс – это программный код, который задает атрибуты данных и методы для объекта определенного типа.

Прежде чем объект будет создан, он должен быть разработан программистом. Программист определяет атрибуты данных, необходимые

методы и затем создает класс. Класс – это программный код, который задает атрибуты данных и методы объекта определенного типа. Представьте класс как «строительный проект», на основе которого будут возводиться объекты.

Класс выполняет аналогичные задачи, что и проект дома. Сам проект не является домом, он выступает подробным описанием дома. Когда для возведения реального дома используется строительный проект, обычно говорят, что типовой дом (экземпляр дома) возводится согласно проекту. По желанию заказчика может быть построено несколько идентичных зданий на основе одного и того же проекта. Каждый возведенный дом является отдельным экземпляром дома, описанного в проекте. Эта идея проиллюстрирована на рис. 10.3.

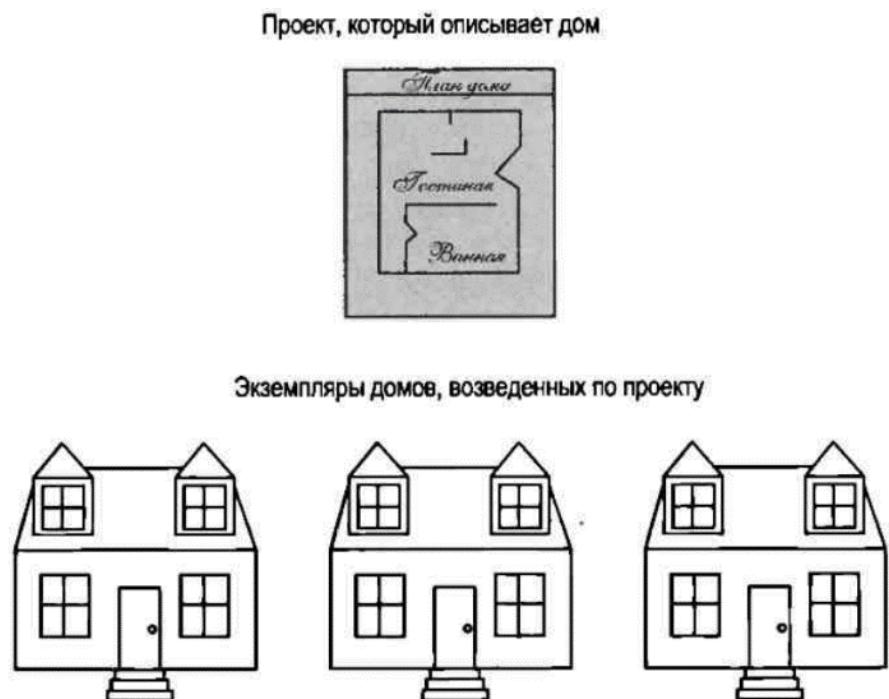


Рисунок 10.3 – Проект дома как класс и возведенные по проекту дома как объекты

Итак, класс – это описание свойств объекта. Когда программа работает, она может использовать класс для создания в оперативной памяти такого количества объектов определенного типа, какое понадобится. Каждый объект, который создается на основе класса, называется экземпляром класса.

Например, энтомолог (специалист, изучающий насекомых), пишет компьютерные программы. Этот человек разрабатывает программу для каталогизации различных типов насекомых. В рамках программы он создает класс **Insect** (Насекомое), который задает свойства, характерные для всех типов насекомых. Класс **Insect** представляет собой спецификацию, согласно которой создаются объекты. Далее она пишет программные инструкции, создающие объект под названием **housefly** (комнатная муха), который является экземпляром класса **Insect**. Объект **housefly** – это сущность, которая занимает место в оперативной памяти компьютера и там хранит данные о комнатной мухе.

Этот объект имеет атрибуты данных и методы, заданные классом **Insect**. Затем программист пишет программные инструкции, которые создают объект под названием **mosquito** (комар). Объект **mosquito** тоже является экземпляром класса **Insect**. Он занимает собственную область в оперативной памяти и там хранит данные о комаре. И хотя объекты **housefly** и **mosquito** в оперативной памяти компьютера являются отдельными объектами, они оба были созданы на основе класса **Insect**. Другими словами, каждый из этих объектов имеет атрибуты данных и методы, описанные классом **Insect**. Это проиллюстрировано на рис. 10.4.

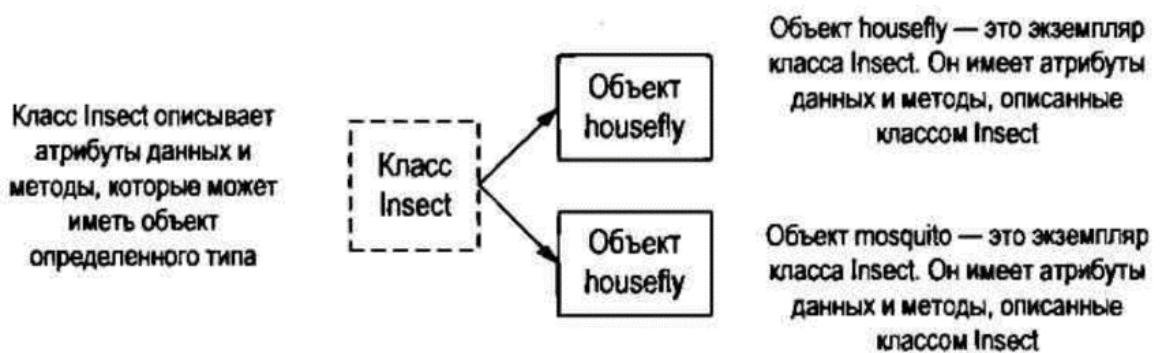


Рисунок 10.4 – Объекты **housefly** и **mosquito** являются экземплярами класса **Insect**

### 10.2.1. Определения классов

Для того чтобы создать класс, пишут определение класса – набор инструкций, которые задают методы класса и атрибуты данных. Давайте взглянем на простой пример. Предположим, что мы пишем программу для имитации бросания монеты. В программе мы должны неоднократно имитировать подбрасывание монеты и всякий раз определять, приземлилась ли она орлом либо решкой. Принимая объектно-ориентированный подход, напишем класс **Coin** (Монета), который может выполнять действия монеты.

В следующей программе представлено определение класса:

```
1 from random import randint
2
3 class Coin:
4 def __init__(self):
5 self.sideup = 'Орёл'
6
7 def toss(self):
8 if randint(0,1) == 0:
9 self.sideup = 'Орёл'
10 else:
11 self.sideup = 'Решка'
12
13 def get_sideup(self):
14 return self.sideup
```

В строке 1 мы импортируем функцию **randint** из модуля **random**, т. к. необходима генерация случайного числа. Стока 3 является началом определения класса. Оно начинается с ключевого слова **class**, за которым идет имя класса, т. е. **Coin**, и потом двоеточие.

К именам классов применимы те же самые правила, которые применимы к именам переменных. Однако следует учесть, что мы начинаем имя класса, **Coin**, с заглавной буквой. Такое написание необязательно. Оно является широко распространенным среди программистов соглашением о наименовании классов. При чтении программного кода такое написание помогает легко отличать имена классов от имен переменных.

Класс **Coin** имеет три метода:

- ◆ метод `__init__( )` появляется в строках 4-5;
- ◆ метод `toss( )` (подбрасывать) – в строках 7-11;
- ◆ метод `get_sideup( )` (получить обращенную вверх сторону монеты) – в строках 13-14.

Обратите внимание, что за исключением того, что эти определения методов появляются в классе, они похожи на любое другое определение функции в Python. Они начинаются со строки заголовка, после которой идет выделенный отступом блок инструкций.

Взгляните на заголовок каждого определения метода (строки 4, 7 и 13) и обратите внимание, что каждый метод имеет параметрическую переменную с именем **self**:

Параметр **self** требуется в каждом методе класса. Метод оперирует атрибутами данных конкретного объекта. Во время исполнения метод должен иметь возможность знать, атрибутами данных какого объекта он призван оперировать. Именно здесь на первый план выходит параметр **self**. Когда метод вызывается, Python делает так, что параметр **self** ссылается на конкретный объект, которым этот метод призван оперировать.

Большинство классов Python имеет специальный метод `__init__( )`, который автоматически исполняется, когда экземпляр класса создается в оперативной памяти. Метод `__init__( )` обычно называется методом инициализации, потому что он инициализирует атрибуты данных объекта. (Название метода состоит из двух символов подчеркивания, слова **init** и еще двух символов подчеркивания.)

Сразу после создания объекта в оперативной памяти исполняется метод `__init__( )`, и параметру **self** автоматически присваивается объект, который был только что создан. Внутри этого метода исполняется инструкция в строке 5:

```
self.sideup = 'Орёл'
```

Эта инструкция присваивает строковый литерал '**Орёл**' атрибуту данных **sideup**, принадлежащему только что созданному объекту. В результате работы метода **\_\_init\_\_( )** каждый объект, который мы создаем на основе класса **Coin**, будет первоначально иметь атрибут **sideup** с заданным значением '**Орёл**'.

Метод **\_\_init\_\_( )** обычно является первым методом в определении класса.

Метод **toss( )** в строках 7-11 тоже имеет необходимую параметрическую переменную **self**. При вызове метода **toss( )** параметрическая переменная **self** автоматически будет ссылаться на объект, которым этот метод должен оперировать.

Метод **toss( )** имитирует подбрасывание монеты. Когда он вызывается, инструкция **if** в строке 8 вызывает функцию **randint** для получения случайного целого числа в диапазоне от 0 до 1. Если число равняется 0, то инструкция в строке 9 присваивает атрибуту **self.sideup** значение '**Орёл**'. В противном случае инструкция в строке 11 присваивает атрибуту **self.sideup** значение '**Решка**'.

Метод **get\_sideup( )** появляется в строках 13-14 просто возвращает значение атрибута **self.sideup**. Данный метод вызывается в любое время, когда возникает необходимость узнать, какой стороной монета обращена вверх.

После подобного определения класса его можно использовать в программе как новый, определенный пользователем, тип данных. То есть создавать переменные типа **Coin**:

```

1 my_coin = Coin()
2
3 while True:
4 print('Бросаю монету...')
5 my_coin.toss()
6
7 print('Монета упала этой стороной вверх: ', my_coin.get_sideup())
8
9 answer = input('Бросить монету еще раз? (д/н) > ')
10 if answer.upper()=='Н':
11 break

```

В данном коде циклически производится «подбрасывание» монеты, то есть вызов функции `toss()`, пока пользователь не введет '`н`' или '`Н`'.

В строке 1 расположена инструкция:

`my_coin = Coin()`

Выражение `Coin()`, которое расположено справа от оператора `=`, приводит к тому, что:

- ◆ в оперативной памяти создается объект на основе класса `Coin`;
- ◆ исполняется метод `__init__()` класса `Coin`, и параметру `self` автоматически назначается объект, который был только что создан. В результате атрибуту `sideup` этого объекта присваивается строковый литерал '`Орёл`'.

На рис. 10.5 проиллюстрированы эти шаги.

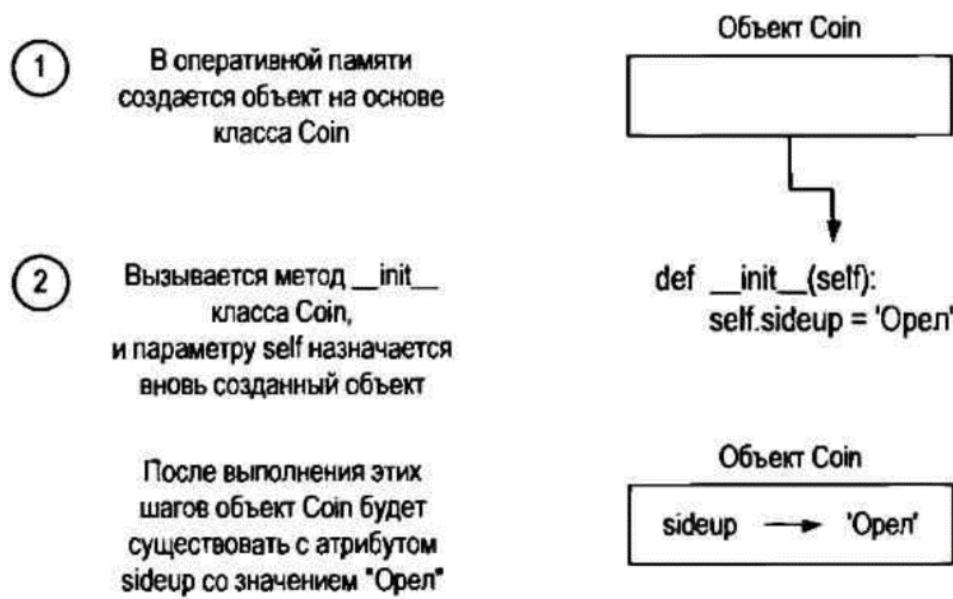


Рисунок 10.5 – Действия, вызванные выражением `Coin()`

После этого оператор `=` присваивает только что созданный объект `Coin` переменной `my_coin`. На рис. 10.6 показано, что после исполнения инструкции в строке 12 переменная `my_coin` будет ссылаться на объект `Coin`, а атрибуту `sideup` этого объекта присвоен строковый литерал '`Орёл`'.

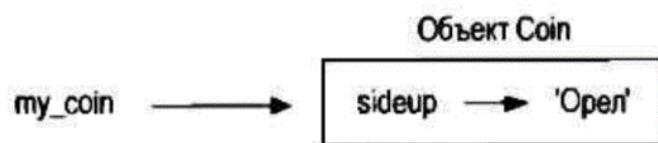


Рисунок 10.6 – Переменная `my_coin` ссылается на объект `Coin`

Вывод данной программы может быть следующим:

```

 ↳ Бросаю монету...
 Монета упала этой стороной вверх: Решка
 Бросить монету еще раз? (д/н) > д
 Бросаю монету...
 Монета упала этой стороной вверх: Орёл
 Бросить монету еще раз? (д/н) > д
 Бросаю монету...
 Монета упала этой стороной вверх: Решка
 Бросить монету еще раз? (д/н) > д
 Бросаю монету...
 Монета упала этой стороной вверх: Решка
 Бросить монету еще раз? (д/н) > д
 Бросаю монету...
 Монета упала этой стороной вверх: Орёл
 Бросить монету еще раз? (д/н) > н

```

### 10.2.2. Скрытие атрибутов

Ранее уже упоминалось, что атрибуты данных объекта должны быть приватными для того, чтобы только методы объекта имели к ним непосредственный доступ. Такой подход защищает атрибуты данных от случайного повреждения. Однако в классе `Coin`, который был показан в предыдущем примере, атрибут `sideup` не является приватным. К нему могут получать непосредственный доступ инструкции, отсутствующие в методе класса `Coin`. Например:

```

1 my_coin = Coin()
2 my_coin.toss()
3 my_coin.sideup = 'Ребро'
4
5 print('Монета упала этой стороной вверх: ', my_coin.get_sideup())

```

▷ Монета упала этой стороной вверх: Ребро

Несмотря на то, что в строке 2 был вызван метод `toss()`, в строке 3 происходит прямое задание (не случайное) значения переменной `sideup`, которая получает вообще недопустимое значение.

Если необходимо сымитировать подбрасывание монеты по-настоящему, надо сделать так, чтобы код за пределами класса не имел возможности менять результат метода `toss()`. А для этого следует сделать атрибут `sideup` приватным. В Python атрибут можно скрыть, если предварить его имя двумя символами подчёркивания. Если изменить имя атрибута `sideup` на `__sideup`, то программный код за пределами класса `Coin` не сможет получать к нему доступ:

```

1 from random import randint
2
3 class Coin:
4 def __init__(self):
5 self.__sideup = 'Орёл'
6
7 def toss(self):
8 if randint(0,1) == 0:
9 self.__sideup = 'Орёл'
10 else:
11 self.__sideup = 'Решка'
12
13 def get_sideup(self):
14 return self.__sideup
15
16 my_coin = Coin()
17 my_coin.toss()
18 my_coin.__sideup = 'Ребро'
19
20 print('Монета упала этой стороной вверх: ', my_coin.get_sideup())

```

▷ Монета упала этой стороной вверх: Орёл

В строке 18 не произошла инициализация скрытого поля.

### 10.3. Объект. Экземпляры класса

Рассмотрим еще один пример решения задачи с использованием классов.

Компания «Беспроводные решения» продает сотовые телефоны и услуги беспроводной связи. Вы работаете программистом в ИТ-отделе компании, и ваша команда разрабатывает программу для учета сведений обо всех сотовых телефонах, которые находятся на складе.

Вам поручили разработать класс, который представляет сотовый телефон. Вот данные, которые должны храниться в качестве атрибутов класса:

- ◆ имя производителя телефона будет присвоено атрибуту `manufact`;
- ◆ номер модели телефона будет присвоен атрибуту `model`;
- ◆ розничная цена телефона будет присвоена атрибуту `retailprice`.

Кроме того, этот класс будет иметь приведенные ниже методы:

- ◆ метод `__init__()` принимает аргументы для производителя, номера модели и розничной цены;
- ◆ метод `set_manufact( )` принимает аргумент для производителя и в случае необходимости позволит изменять значение атрибута `manufact` после создания объекта;
- ◆ метод `set_model( )` принимает аргумент для модели и в случае необходимости позволит изменять значение атрибута `model` после создания объекта;
- ◆ метод `set_retail_price( )` принимает аргумент для розничной цены и в случае необходимости позволяет изменять значение атрибута `retail_price` после создания объекта;
- ◆ метод `get_manufact( )` возвращает название производителя телефона;
- ◆ метод `get_model( )` возвращает номер модели телефона;
- ◆ метод `get_retail_price( )` возвращает розничную цену телефона.

В программе (рис. 10.7) показано определение класса **CellPhone**, который сохранен в модуле **cellphone**

```

1 class CellPhone:
2 def __init__(self, manufact, model, price):
3 self.manufact = manufact
4 self.model = model
5 self.retail_price = price
6
7 def set_manufact(self, manufact):
8 self.manufact = manufact
9
10 def set_model(self, model):
11 self.model = model
12
13 def set_retail_price(self, price):
14 self.retail_price = price
15
16 def get_manufact(self):
17 return self.manufact
18
19 def get_model(self):
20 return self.model
21
22 def get_retail_price(self):
23 return self.retail_price
24

```

Рисунок 10.7 – Класс **CellPhone**

### 10.3.1. Работа с экземплярами

Рассмотрим пример программы с использованием класса **CellPhone**:

```

▶ 1 c1 = CellPhone("Acme Electronics", "M1000", 200.56)
 2 c2 = CellPhone("Acme Electronics", "SX20", 10.00)
 3 c3 = CellPhone("Atlantic Communicztons", "N47", 150.00)
 4
 5 print(c1)
 6 print(c2)
 7 print(c3)

```

```

8
9 print(f'> Производитель: {c1.get_manufact()},\n\
10 Модель: {c1.get_model()},\n\
11 Розничная цена: {c1.get_retail_price()}')
12
13 print(f'> Производитель: {c2.get_manufact()},\n\
14 Модель: {c2.get_model()},\n\
15 Розничная цена: {c2.get_retail_price()}')
16
17 print(f'> Производитель: {c3.get_manufact()},\n\
18 Модель: {c3.get_model()},\n\
19 Розничная цена: {c3.get_retail_price()}'')

```

Рисунок 10.8 – Класс **CellPhone**

Вывод данной программы:

```

<__main__.CellPhone object at 0x7b60c0e14ee0>
<__main__.CellPhone object at 0x7b60c0e15990>
<__main__.CellPhone object at 0x7b60c0e16b30>
> Производитель: Acme Electronics,
Модель: M1000,
Розничная цена: 200.56
> Производитель: Acme Electronics,
Модель: SX20,
Розничная цена: 10.0
> Производитель: Atlantic Communicztons,
Модель: N47,
Розничная цена: 150.0

```

Каждый экземпляр класса имеет собственный набор атрибутов данных.

При использовании методом параметра **self** для создания атрибута этот атрибут принадлежит конкретному объекту, на который ссылается параметр **self**. Мы называем такие атрибуты атрибутами экземпляра, потому что они принадлежат конкретному экземпляру класса.

В программе можно создавать многочисленные экземпляры одного и того же класса. И каждый экземпляр будет иметь собственный набор атрибутов. Например в программе на рис. 10.8 создаются три экземпляра класса **CellPhone**.

### 10.3.2. Методы-получатели и методы-мутаторы

На практике широко принято делать в классе все атрибуты данных приватными и предоставлять публичные методы для доступа к этим атрибутам и их изменения. Так гарантируется, что объект, владеющий этими атрибутами, будет держать под контролем все вносимые в них изменения.

Метод, который возвращает значение из атрибута класса и при этом его не изменяет, называется методом-получателем. Методы-получатели дают возможность программному коду, находящемуся за пределами класса, извлекать значения атрибутов безопасным образом, не подвергая эти атрибуты изменению программным кодом, находящимся вне метода. В классе **CellPhone**, который вы увидели в программе 10.7, методы `get_manufact()`, `get_model()` и `get_retail_price()` являются методами-получателями.

Метод, который сохраняет значение в атрибуте данных либо каким-нибудь иным образом изменяет значение атрибута данных, называется методом-мутатором. Методы-мутаторы могут управлять тем, как атрибуты данных класса изменяются. Когда программный код, находящийся вне класса, должен изменить в объекте значение атрибута данных, он, как правило, вызывает мутатор и передает новое значение в качестве аргумента. Если это необходимо, то мутатор, прежде чем он присвоит значение атрибуту данных, может выполнить проверку этого значения. В программе 10.7 методы `set_manufact()`, `set_model()` и `set_retail_price()` являются методами-мутаторами.

Методы-мутаторы иногда называют сеттерами (setter), или методами-установщиками, а методы-поггучатели – геттерами (getter)

### 10.3.3. Метод `__str__`

Важное следствие из кода на рис. 10.8: объект класса, определенного пользователем нельзя напечатать функцией `print` с выводом значений атрибутов, то есть внутреннего состояния объекта (строки 5-7). Вместо этого

приходится обращаться к методам-получателям каждого объекта (строки 9-19). В результате получается многократное повторение практически одинакового кода (так называемый макаронный код).

Вывод на экран состояния объекта – широко распространенная задача. Причем настолько, что многие программисты оснащают свои классы методом, который возвращает строковое значение, содержащее состояние объекта. В Python этому методу присвоено специальное имя `__str__`.

Метод `__str__()` вызывается не напрямую, а автоматически во время передачи объекта в качестве аргумента в функцию `print`. Метод `__str__()` определим следующим образом (строки 25-28) внутри класса `CellPhone`:

```

1 class CellPhone:
2 def __init__(self, manufact, model, price):
3 self.manufact = manufact
4 self.model = model
5 self.retail_price = price
6
7 def set_manufact(self, manufact):
8 self.manufact = manufact
9
10 def set_model(self, model):
11 self.model = model
12
13 def set_retail_price(self, price):
14 self.retail_price = price
15
16 def get_manufact(self):
17 return self.manufact
18
19 def get_model(self):
20 return self.model
21
22 def get_retail_price(self):
23 return self.retail_price
24
25 def __str__(self):
26 return '*'*20 + f'\nПроизводитель: {c2.get_manufact()},{\n' +\
27 f'Модель: {c2.get_model()},{\n' +\
28 f'Розничная цена: {c2.get_retail_price()}\n' + '*'*20

```

Рисунок 10.9 – Класс `CellPhone` с методом `__str__()`

Теперь можно использовать объекты типа `CellPhone` в качестве параметров функции `print`:

```

1 c1 = CellPhone("Acme Electronics", "M1000", 200.56)
2 c2 = CellPhone("Acme Electronics", "SX20", 10.00)
3 c3 = CellPhone("Atlantic Communiczctions", "N47", 150.00)
4
5 print(c1)
6 print(c2)
7 print(c3)

```

```

Производитель: Acme Electronics,
Модель: SX20,
Розничная цена: 10.0

Производитель: Acme Electronics,
Модель: SX20,
Розничная цена: 10.0

Производитель: Acme Electronics,
Модель: SX20,
Розничная цена: 10.0

```

Если необходимо выводить телефоны в другом представлении, необходимо только изменить код в одном месте – в функции `__str__()` класса `CellPhone`.

#### 10.4. Консервация пользовательских объектов

Ранее в курсе (пп. 8.4) было рассмотрено, что модуль `pickle` предоставляет функции для сериализации объектов. Сериализация объекта означает его преобразование в поток байтов, которые могут быть сохранены в файле для последующего извлечения. Функция `dump` модуля `pickle` сериализует (консервирует) объект и записывает его в файл, а функция `load` извлекает объект из файла и его десериализует (расконсервирует).

Пользовательские объекты также сериализуются средствами `pickle`. Консервирование объекта показано в следующем коде:

```

1 import pickle
2 c1 = CellPhone("Acme Electronics", "M1000", 200.56)
3 c2 = CellPhone("Acme Electronics", "SX20", 10.00)
4 c3 = CellPhone("Atlantic Communiczctions", "N47", 150.00)
5
6 output_file = open ('data.dat', 'wb')
7 pickle.dump(c1, output_file)
8 pickle.dump(c2, output_file)
9 pickle.dump(c3, output_file)
10 output_file.close()

```

Следующий листинг демонстрирует расконсервацию объекта:

```
1 import pickle
2 input_file = open ('data.dat', 'rb')
3 end_of_file = False
4 while not end_of_file:
5 try:
6 phone = pickle.load(input_file)
7 print(phone)
8 except:
9 end_of_file = True
```

```

Производитель: Acme Electronics,
Модель: SX20,
Розничная цена: 10.0

Производитель: Acme Electronics,
Модель: SX20,
Розничная цена: 10.0

Производитель: Acme Electronics,
Модель: SX20,
Розничная цена: 10.0

```

В данном примере каждый десериализуемый объект выводится на печать, но можно реализовать добавление объектов в список для дальнейшей их обработки.

### **Вопросы для самопроверки по теме 10**

1. Что такое объект?
2. Что такое инкапсуляция?
3. Почему внутренние данные объекта обычно скрыты от внешнего программного кода?
4. Что такое публичные методы? Что такое приватные методы?
5. Вы слышите, что кто-то высказывает следующий комментарий: «Проект – это дизайн дома. Плотник использует проект для возведения дома. Если плотник пожелает, он может построить несколько идентичных домов на основе одного и того же проекта». Представьте это как метафору для классов и объектов. Этот проект представляет класс или же он представляет объект?
6. Какова задача метода `__init__( )`? И когда он исполняется?

7. Какова задача параметра `self` в методе?
8. Каким образом в классе Python атрибут скрывается от программного кода, находящегося за пределами класса?
9. Какова задача метода `__str__( )`?
10. Каким образом происходит вызов метода `__str__( )`?
11. Что такое атрибут экземпляра?
12. Что такое метод-получатель? Что такое метод-мутатор?

## Лекция 11. Анонимные функции

План лекции

1. Лямбда-выражения.
2. Применение анонимных функций в алгоритмах сортировки, фильтрации и отображения.

### 11.1. Лямбда-выражения

Во многих языках программирования существует такое явление как анонимные функции. Данные функции также называются лямбда-выражениями. Рассмотрим пример объявления и вызова простой функции:

```
 1 def f(x):
2 return x*x
3
4 print(f(4))
```

→ 16

Рисунок 11.1 – Пример простой функции

Создавать отдельную функцию для такого простого действия не имело смысла. В таких случаях можно создать анонимную функцию – без названия и отдельного определения с использованием **def**. Анонимная (неименованная) функция может определяться лямбда-выражением:

**lambda аргументы: выражение**

где **lambda** – зарезервированное слово; **аргументы** – список аргументов; **выражение** – тело функции, то что функция вернет с использованием **return**.

Вместо функции на рис. 11.1 можно использовать лямбда-выражение:

```
 1 p = lambda x : x*x
2
3 print(p(5))
4 print(p(10))
```

→ 25  
100

Рисунок 11.2 – Пример лямбда-выражения

Код, определяемый с ключевым словом `lambda`, должен быть действительным выражением. Лямбда-функции не могут занимать более одной строки, и в них не могут использоваться команды, не являющиеся выражениями (`try` и `while`).

## 11.2. Применение анонимных функций в алгоритмах сортировки, фильтрации и отображения

### 11.2.1. Применение лямбда-выражений для сортировки

Одно из главных применений `lambda` – определение небольших функций обратного вызова. Например, лямбда-функции нередко встречаются при использовании со встроенными операциями, такими как сортировка. Рассмотрим пример сортировки списка:

```
 1 data = [1,45, 6, 324, 1056, 67, 123]
 2 data.sort()
 3 print(data)

→ [1, 6, 45, 67, 123, 324, 1056]
```

Рисунок 11.3 – Использование метода `sort()`

Возможно также выполнить сортировку в обратном порядке:

```
 1 data = [1,45, 6, 324, 1056, 67, 123]
 2 data.sort(reverse=True)
 3 print(data)

[1056, 324, 123, 67, 45, 6, 1]
```

Но что делать, если необходимо выполнить более сложную сортировку, например по сумме цифр числа: сначала вывести числа с минимальной суммой цифр. В следующем примере показана сортировка с использованием функции `my_sort()`, которая используется в качестве значения параметра `key` метода `sort()`:

```

1 def my_sort(x):
2 s = 0
3 sx = str(x)
4 for c in sx:
5 s += int(c)
6 return s
7
8 data = [1,45, 6, 324, 1056, 67, 123]
9 data.sort(key=my_sort)
10 print(data)

```

⇒ [1, 6, 123, 45, 324, 1056, 67]

Рисунок 11.4 – Использование параметра **key** в методе **sort()**

В 11.4 в качестве параметра **key** метода **sort()** указывается имя функции **my\_sort()**, которая будет использована для сортировки. Данная функция имеет следующую структуру: в качестве параметра предполагается получение самого элемента сортируемого списка, а в качестве возвращаемого значения – числовое значение на основе которого будет произведена сортировка. Таким образом в методе **sort()** для каждого значения списка производится вычисление функции **my\_sort()**, то есть расчет суммы цифр и сортировка производится на основе полученных сумм.

Пример 11.4 можно реализовать с использованием лямбда-выражений:

```

1 data = [1,45, 6, 324, 1056, 67, 123]
2 data.sort(key=lambda x : sum(int(c) for c in str(x)))
3 print(data)

```

⇒ [1, 6, 123, 45, 324, 1056, 67]

Рисунок 11.5 – Использование параметра **key** с лямбда-выражением в методе **sort()**

### 11.2.2. Применение лямбда-выражений для фильтрации

Рассмотрим пример использования лямбда выражений для фильтрации (отбора) элементов списка по определенному критерию. Например, необходимо из списка отобрать элементы, кратные 3:

```
1 data = [1,45, 6, 324, 1056, 67, 123]
2 data3 = filter(lambda x : x%3, data)
3 print(data3)
4 print(type(data3))
```

```
<filter object at 0x7c07e80c89d0>
<class 'filter'>
```

Рисунок 11.6 – Использование параметра **key** с лямбда-выражением в функции **filter()**

В данном примере мы произвели фильтрацию, но функция `filter()` вернула не список подходящих значений, а объект типа `<class 'filter'>`. Для получения требуемого результата необходимо выполнить преобразование к типу `list` (в общем случае к любому требуемому типу):

```
1 data = [1,45, 6, 324, 1056, 67, 123]
2 data3 = list(filter(lambda x : x%3, data))
3 print(data3)
```

```
⇒ [1, 67]
```

Рисунок 11.7 – Использование параметра **key** с лямбда-выражением в функции **filter()**

На самом деле получить аналогичный результат можно было с использованием инициализаторов списка:

```
1 data = [1, 45, 6, 324, 1056, 67, 123]
2 data3 = [i for i in data if i%3==0]
3 print(data3)
```

```
⇒ [45, 6, 324, 1056, 123]
```

### 11.2.3. Применение лямбда-выражений для отображения

Аналогично использованию функции `filter` можно осуществить маппинг (отображение) элементов одного списка в другой. Рассмотрим пример.

Пусть необходимо на основе списка, содержащего значения различных типов получить список квадратов этих элементов. Если для элемента невозможно вычислить квадрат, то для него необходимо вернуть значение `-1`.

Пример реализации:

```
1 def my_map(x):
2 if type(x) in [int, float]:
3 return x*x
4 else:
5 return -1
6
7 data = [34, 45.67, 'Hel', 90, [23, 0], "abc", 45]
8 map_object = map(my_map, data)
9 print(map_object)
10 print(type(map_object))
11 map_data = list(map_object)
12 print(map_data)

⇒ <map object at 0x7c07dbdef100>
<class 'map'>
[1156, 2085.7489, -1, 8100, -1, -1, 2025]
```

Отображение производится с использованием функции `map()`, которая имеет следующий формат:

**map(функция, коллекция)**

где **функция** – это определенная заранее функция, задающая преобразование, которое необходимо применить к каждому элементу коллекции **коллекция**.

Функция `map()` не возвращает готовую коллекцию, она возвращает объект типа `<class 'map'>`. Для получения отображеного списка необходимо выполнить преобразование к типу `list`.

Предыдущий пример можно было реализовать с использованием лямбда выражений:

```
1 data = [34, 45.67, 'Hel', 90, [23, 0], "abc", 45]
2 map_data = list(map(lambda x : \
3 x*x if type(x) in [int, float] \
4 else -1, data))
5 print(map_data)

⇒ [1156, 2085.7489, -1, 8100, -1, -1, 2025]
```

Рассмотрим распространенную операцию маппинга, когда из единой строки необходимо выделить числа и получить список из них:

```
1 sx = '123 45 67 89 12 1 4'
2 x = list(map(int, sx.split()))
3 print(x)

⇒ [123, 45, 67, 89, 12, 1, 4]
```

**Вопросы для самопроверки по теме 11**

1. Что такое анонимная функция?
2. Опишите синтаксис объявления лямбда-выражения.
3. Поясните назначение лямбда-выражений. Приведите примеры их использования.

4. Приведите пример использования лямбда выражений для фильтрации элементов списка.

5. Что будет напечатано в результате выполнения кода:

```
sx = '1.1 3.3 6.6'
x = list(map(float, sx.split()))
print(x[1])
```

5. Что будет напечатано в результате выполнения кода:

```
data = [13, 1.3, 4, 'abs', 78]
filtered_data = list(filter(lambda x: type(x)==int, data))
print(filtered_data)
```

## Лекция 12. Визуализация данных

### План лекции

1. Манипулирование данными средствами pandas
2. Построение графиков и диаграмм.

### 12.1. Манипулирование данными средствами pandas

Задачи визуализации рассмотрим на основе набора данных «Выпускники ВУЗов». Данный набор данных используется в качестве учебного набора данных. Набор представляет собой данные о выпускниках американских вузов в разрезе направлений подготовки. Набор данных можно получить по ссылке:

[https://github.com/enikolaev/Algorithmization\\_and\\_programming/blob/main/data/recent-grads-test.csv](https://github.com/enikolaev/Algorithmization_and_programming/blob/main/data/recent-grads-test.csv)

Конечно, так как набор данных содержится в текстовом файле CSV можно использовать встроенные возможности Python по работе с текстовыми файлами, но в экосистеме Python существует более эффективная библиотека для работы с табличными наборами данных. Эта библиотека называется **pandas**.

Рассмотрим основные признаки, представленные в наборе данных. Загрузим набор данных на сервер (рисунок 12.1).

```
[1] 1 # Выполним магическую команду
2 # и загрузим данные с github
3 !wget https://raw.githubusercontent.com/enikolaev/Algorithmization_and_programming/main/data/recent-grads-test.csv
```

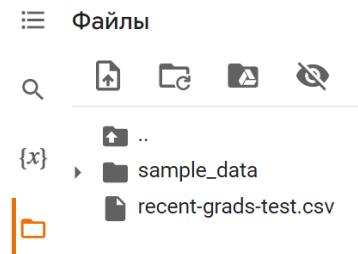
--2023-09-05 11:00:01-- [https://raw.githubusercontent.com/enikolaev/Algorithmization\\_and\\_programming/main/data/recent-grads-test.csv](https://raw.githubusercontent.com/enikolaev/Algorithmization_and_programming/main/data/recent-grads-test.csv)
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.152
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.152|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 19321 (19K) [text/plain]
Saving to: ‘recent-grads-test.csv’

recent-grads-test.csv 100%[=====] 18.87K --.-KB/s in 0s

2023-09-05 11:00:02 (67.7 MB/s) - ‘recent-grads-test.csv’ saved [19321/19321]

Рисунок 12.1 – Загрузка данных

Очевидно, что файл **recent-grads-test.csv.csv** можно просто загрузить в рабочую папку Google Colab без использования магических команд. После загрузки файла, он появится в рабочем каталоге:



Теперь можно воспользоваться библиотекой **pandas** для чтения данных из файла и сохранения всего массива данных в специальном типе данных **DataFrame** (рисунок 12.2).

```
▶ 1 import pandas as pd
 2 data = pd.read_csv('recent-grads-test.csv', delimiter=';')
 3 data.head()
```

Рисунок 12.2 – Считывание набора данных из файла в **DataFrame**

Результат выполнения представленного кода:

| rank | Major_code | Major                                     | Total   | Men     | Women   | Major_category | Employed | Full_time | Part_time | Unemployed | Median earning | College_jobs | Non_co |
|------|------------|-------------------------------------------|---------|---------|---------|----------------|----------|-----------|-----------|------------|----------------|--------------|--------|
| 1    | 2419       | PETROLEUM ENGINEERING                     | 2339.0  | 2057.0  | 282.0   | Engineering    | 1976     | 1849      | 270       | 37         | 110000         | 1534         |        |
| 2    | 2416       | MINING AND MINERAL ENGINEERING            | 756.0   | 679.0   | 77.0    | Engineering    | 640      | 556       | 170       | 85         | 75000          | 350          |        |
| 3    | 2415       | METALLURGICAL ENGINEERING                 | 856.0   | 725.0   | 131.0   | Engineering    | 648      | 558       | 133       | 16         | 73000          | 456          |        |
| 4    | 2417       | NAVAL ARCHITECTURE AND MARINE ENGINEERING | 1258.0  | 1123.0  | 135.0   | Engineering    | 758      | 1069      | 150       | 40         | 70000          | 529          |        |
| 5    | 2405       | CHEMICAL ENGINEERING                      | 32260.0 | 21239.0 | 11021.0 | Engineering    | 25694    | 23170     | 5180      | 1672       | 65000          | 18314        |        |

Функция **head( )** выводить только несколько первых записей из объекта **data** типа **DataFrame**. В следующем фрагменте кода вызывается метод **info( )** объекта **data**. Данный метод выводит краткую статистику по набору данных: имена полей, количество непустых значений в каждом столбце и тип данных в каждом столбце.

 1 data.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 173 entries, 0 to 172
Data columns (total 15 columns):
 # Column Non-Null Count Dtype

 0 Rank 173 non-null int64
 1 Major_code 173 non-null int64
 2 Major 173 non-null object
 3 Total 172 non-null float64
 4 Men 172 non-null float64
 5 Women 172 non-null float64
 6 Major_category 173 non-null object
 7 Employed 173 non-null int64
 8 Full_time 173 non-null int64
 9 Part_time 173 non-null int64
 10 Unemployed 173 non-null int64
 11 Median earning 173 non-null int64
 12 College_jobs 173 non-null int64
 13 Non_college_jobs 173 non-null int64
 14 Low_wage_jobs 173 non-null int64
dtypes: float64(3), int64(10), object(2)
memory usage: 20.4+ KB
```

В таблице 12.1 приводится расшифровка полей набора данных:

Таблица 12.1 – Значение полей в наборе данных

| Имя поля в наборе       | Описание                                              |
|-------------------------|-------------------------------------------------------|
| <b>Rank</b>             | Рейтинг специальности                                 |
| <b>Major_code</b>       | Код специальности                                     |
| <b>Major</b>            | Специальность                                         |
| <b>Total</b>            | Всего выпускников                                     |
| <b>Men</b>              | Выпускников мужского пола                             |
| <b>Women</b>            | Выпускников женского пола                             |
| <b>Major_category</b>   | Группа специальности                                  |
| <b>Employed</b>         | Трудоустроенные по специальности                      |
| <b>Full_time</b>        | Трудоустроенные на полный рабочий день                |
| <b>Part_time</b>        | Трудоустроенные с частичной занятостью                |
| <b>Unemployed</b>       | Безработные выпускники                                |
| <b>Median earning</b>   | Медианная заработка выпускников                       |
| <b>College_jobs</b>     | Работают на позициях, требующих высшее образование    |
| <b>Non_college_jobs</b> | Работают на позициях, не требующих высшее образование |
| <b>Low_wage_jobs</b>    | Работают на низкооплачиваемых позициях                |

После того как становится понятным назначение полей, рассмотрим некоторые функции, которые помогут при манипулировании данными.

При необходимости получить данные только одного столбца необходимо воспользоваться операцией индексации с указанием в качестве индекса имени столбца:

```
1 all_majors = data['Major']
2 print(type(all_majors))
3 all_majors.tail()
```

⇨ <class 'pandas.core.series.Series'>
168 ZOOLOGY
169 EDUCATIONAL PSYCHOLOGY
170 CLINICAL PSYCHOLOGY
171 COUNSELING PSYCHOLOGY
172 LIBRARY SCIENCE
Name: Major, dtype: object

В данном коде мы получаем данные только по столбцу '**Major**' объекта **data**. Возвращаемый объект сохраняем в **all\_majors**. Выясняем, что тип переменной **all\_majors** – **Series** (ряд или серия данных). Для объекта типа **Series** также можно выводить порцию данных с использованием **head( )**, но в данном примере используется метод **tail( )**, которые возвращает не первые, а последние **N** записей набора.

Еще один метод, который позволяет вывести статистику по набору данных **describe( )**:

```
1 data.describe()
```

⇨

|       | Rank       | Major_code  | Total         | Men           | Women         | Employed      | Full_time     | Part_time     | Unemployed   |
|-------|------------|-------------|---------------|---------------|---------------|---------------|---------------|---------------|--------------|
| count | 173.000000 | 173.000000  | 172.000000    | 172.000000    | 172.000000    | 173.000000    | 173.000000    | 173.000000    | 173.000000   |
| mean  | 87.000000  | 3879.815029 | 39370.081395  | 16723.406977  | 22646.674419  | 31192.763006  | 26029.306358  | 8832.398844   | 2416.329480  |
| std   | 50.084928  | 1687.753140 | 63483.491009  | 28122.433474  | 41057.330740  | 50675.002241  | 42869.655092  | 14648.179473  | 4112.803148  |
| min   | 1.000000   | 1100.000000 | 124.000000    | 119.000000    | 0.000000      | 0.000000      | 111.000000    | 0.000000      | 0.000000     |
| 25%   | 44.000000  | 2403.000000 | 4549.750000   | 2177.500000   | 1778.250000   | 3608.000000   | 3154.000000   | 1030.000000   | 304.000000   |
| 50%   | 87.000000  | 3608.000000 | 15104.000000  | 5434.000000   | 8386.500000   | 11797.000000  | 10048.000000  | 3299.000000   | 893.000000   |
| 75%   | 130.000000 | 5503.000000 | 38909.750000  | 14631.000000  | 22553.750000  | 31433.000000  | 25147.000000  | 9948.000000   | 2393.000000  |
| max   | 173.000000 | 6403.000000 | 393735.000000 | 173809.000000 | 307087.000000 | 307933.000000 | 251540.000000 | 115172.000000 | 28169.000000 |

Данный метод выводит подробную статистику по набору данных.

Рассмотрим пример, когда программисту необходимо получить максимальное и минимальное значение поля в `DataFrame`, не используя статистику из `describe()`:

```
1 salary = data['Median earning']
2 print(f'Максимальный заработок - {max(salary)}')
3 print(f'Минимальный заработок - {salary.min()}')
```

→ Максимальный заработок - 110000  
Минимальный заработок - 22000

Следует обратить внимание, что вычисление максимального и минимального значений по полю `'Median earning'` производится в представленном листинге различными способами.

## 12.2. Построение графиков и диаграмм

Графики, используемые при анализе данных, делят не по библиотекам, с использованием которых они строятся, а по типам признаков, для анализа которых предназначены графики. Для построения графиков будем использовать библиотеки `matplotlib` и `seaborn`.

### 12.2.1. Визуализация количественных признаков

Для представления распределения простого количественного признака подходит обычная гистограмма, содержащаяся во всех библиотеках (рисунок 12.3).

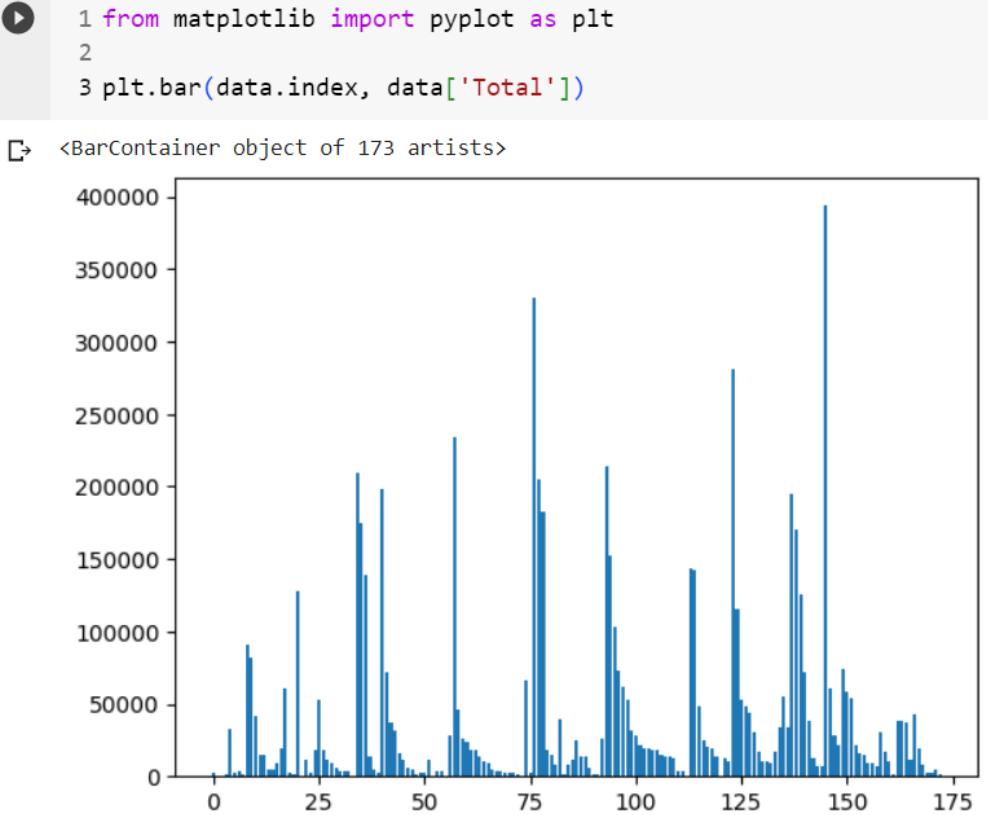


Рисунок 12.3 – Информация о признаках набора данных

Для построения гистограммы вызывается метод `bar()` класса `pyplot`. В качестве параметров передаются два: значения для оси Ох и значения для построения гистограммы. В качестве значений для оси Ох передается `data.index`, то есть внутренний номер (начинается с 0) строки в наборе данных. В качестве значений гистограммы передается `data['Total']`, то есть столбец – суммарное количество выпускников.

Рассмотрим еще одну гистограмму, которая будет выводить не индексы, а наименование специальностей:

```

1 data_salary = data[['Median earning', 'Major']][0:10]
2 print(data_salary)
3 plt.barh(data_salary['Major'], data_salary['Median earning'])

```

В результате выполнения данного кода будет создан набор данных, содержащий два столбца 'Median earning' и 'Major', и только первые 10 записей (строка 1).

Затем полученный набор данных выводится на печать:

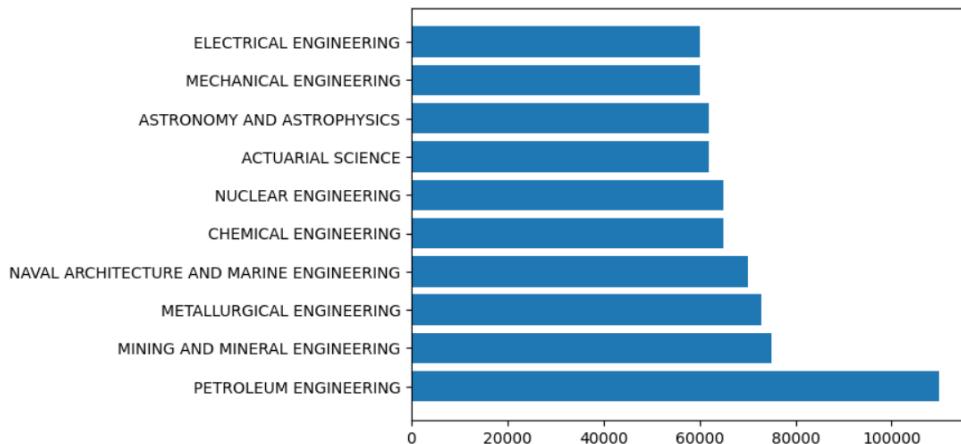
```

 Median earning Major
0 110000 PETROLEUM ENGINEERING
1 75000 MINING AND MINERAL ENGINEERING
2 73000 METALLURGICAL ENGINEERING
3 70000 NAVAL ARCHITECTURE AND MARINE ENGINEERING
4 65000 CHEMICAL ENGINEERING
5 65000 NUCLEAR ENGINEERING
6 62000 ACTUARIAL SCIENCE
7 62000 ASTRONOMY AND ASTROPHYSICS
8 60000 MECHANICAL ENGINEERING
9 60000 ELECTRICAL ENGINEERING

```

<BarContainer object of 10 artists>

И строится гистограмма:



Один из эффективных типов графиков для анализа количественных признаков – это «ящик с усами» (**boxplot**). На рисунке 12.4 показан код и реализованный график. Для анализа нескольких признаков графики **boxplot** также эффективны.

```

1 import seaborn as sns
2 sns.boxplot(y=data['Median earning'], width=0.3)

```

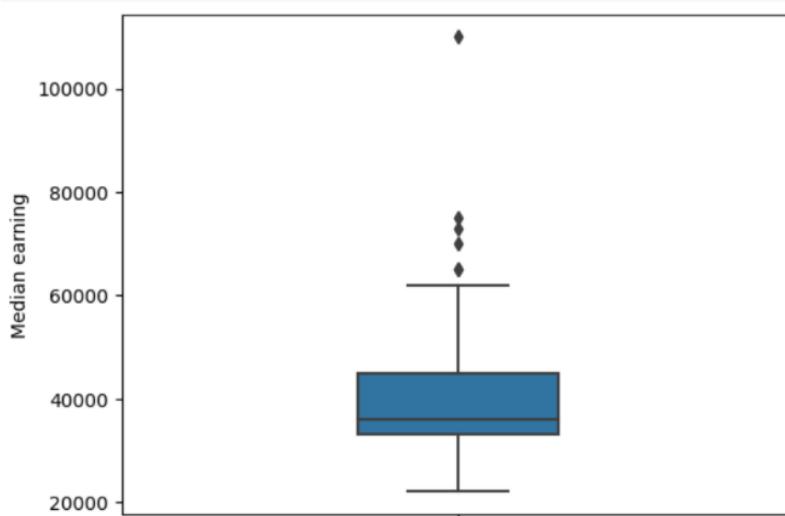


Рисунок 12.4 – График «ящик с усами» для отдельного признака

На рисунке 12.5 представлен код и результат построения графиков для анализа пяти групп специальностей с максимальным уровнем трудоустройства.

```

1 top_data = data[['State','Total day minutes']]
2 top_data = top_data.groupby('State').sum()
3 top_data = top_data.sort_values('Total day minutes', ascending=False)
4 top_data = top_data[:5].index.values
5 sns.boxplot(y='State',
6 x='Total day minutes',
7 data=data[data.State.isin(top_data)], palette='Set3');
```

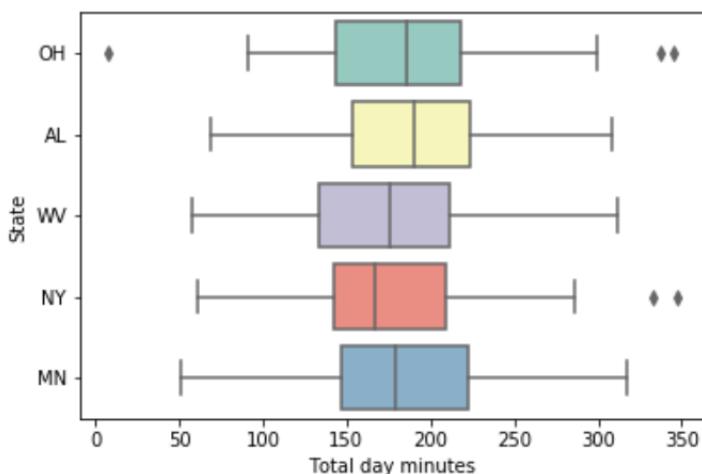


Рисунок 2.5 – Использование boxplot для анализа признака для пяти штатов

График boxplot состоит из коробки, усов и точек. Коробка показывает интерквартильный размах распределения, то есть соответственно 25% (первая квартиль,  $Q_1$ ) и 75% ( $Q_3$ ) перцентили. Черта внутри коробки обозначает медиану распределения (можно получить с использованием метода `median()` в `pandas` и `numpy`). Усы отображают весь разброс точек кроме выбросов, то есть минимальные и максимальные значения, которые попадают в промежуток ( $Q_1 - 1,5 \cdot IQR$ ,  $Q_3 + 1,5 \cdot IQR$ ), где  $IQR = Q_3 - Q_1$  – интерквартильный размах. Точками на графике обозначаются выбросы (outliers), то есть те значения, которые не вписываются в промежуток значений, заданный усами графика (рисунок 2.6).

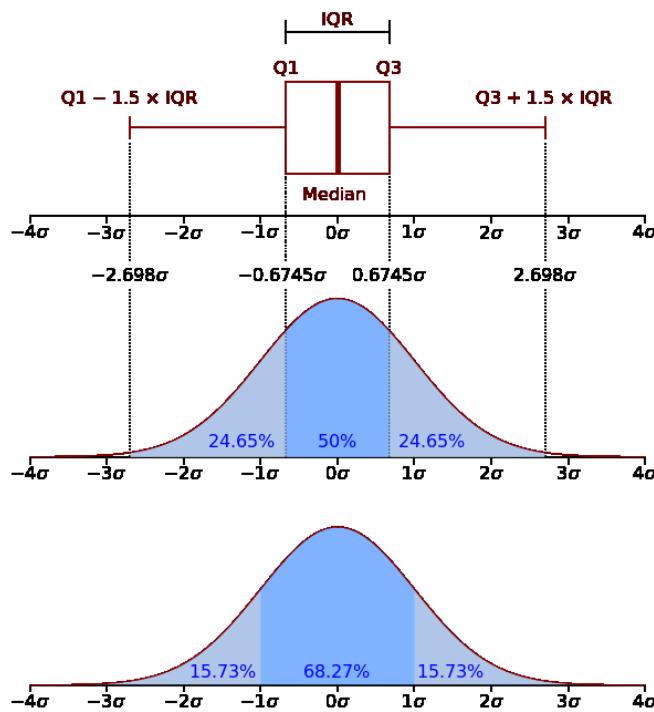


Рисунок 2.6 – Структура графика типа «ящик с усами»

### 12.2.2. Категориальные признаки

Типичным категориальным признаком в анализируемом наборе данных является «Группа специальностей» (`Major_Category`). На рисунке 2.7 представлен график типа `countplot()` из библиотеки `seaborn`, который демонстрирует гистограмму, но не по сырым данным, а по рассчитанному количеству разных значений признака.

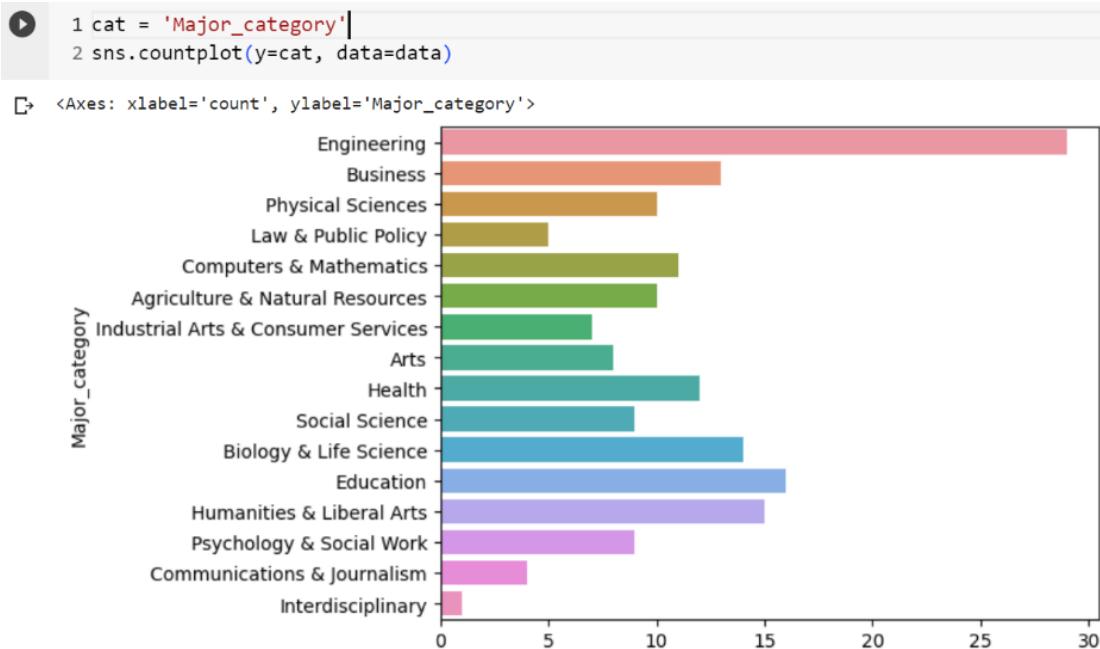


Рисунок 2.7 – Использование **countplot** для анализа категориального признака

### 12.2.3. Визуализация соотношения количественных признаков

Одним из вариантов визуализации соотношения количественных признаков является диаграмма по нескольким признакам. Рассмотрим пример демонстрирующий сравнение распределений показателей, связанных с характером занятости.

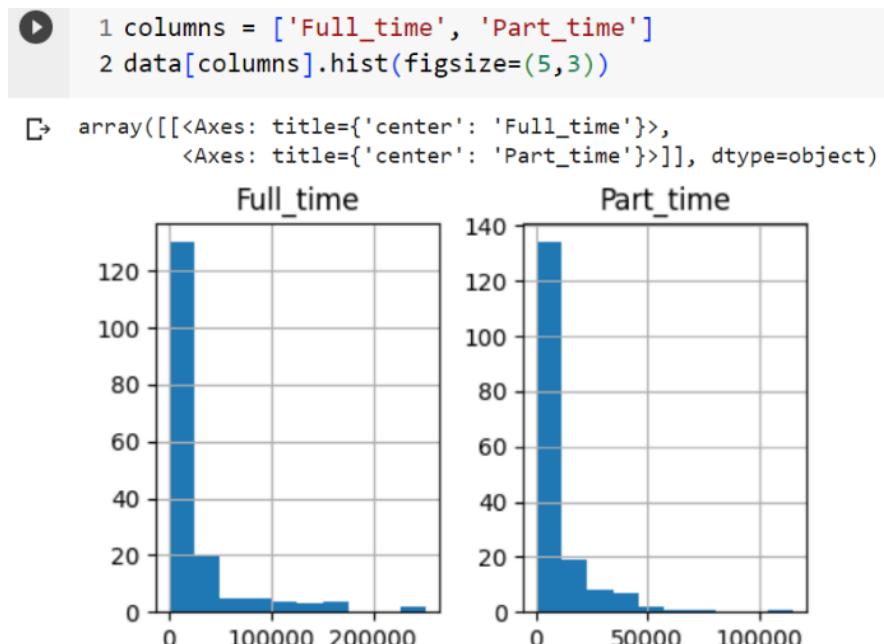


Рисунок 2.8 – Распределения выпускников с частичной и полной занятостью

Часто используют попарное сравнение признаков для обеспечения широкого взгляда на набор данных (рисунок 2.9). На диагональных графиках рисунка 2.9 представлены гистограммы распределения отдельного признака, на внедиагональных позициях – попарные распределения.

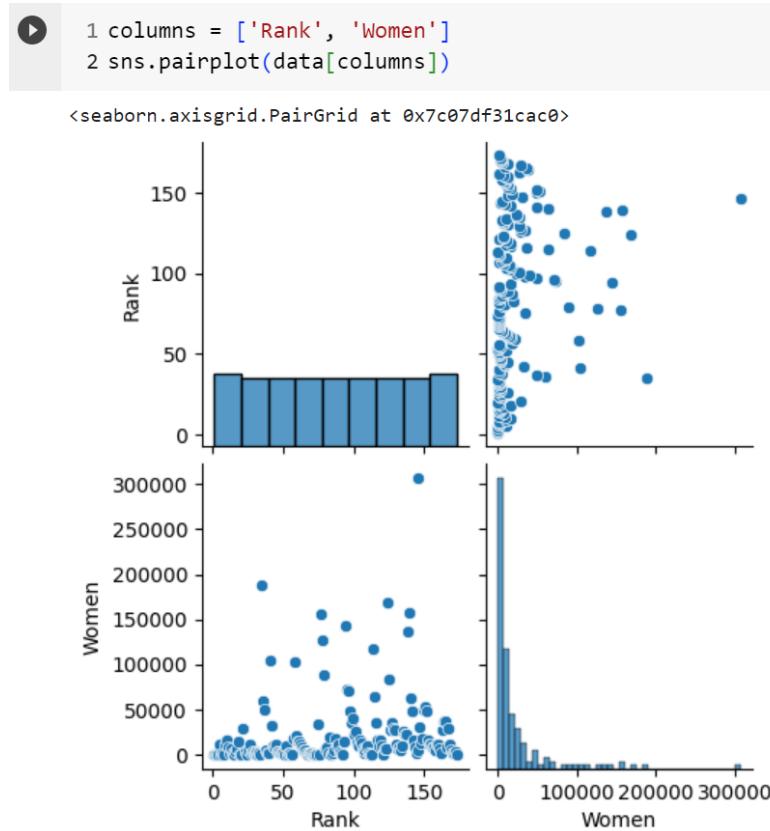


Рисунок 2.9 – Попарное распределение признаков

Попарное распределение, представленное на рис. 2.9, не совсем информативно представляет зависимость. На рис. 2.10 показан более информативный пример. Анализируется попарное распределение признаков '`Unemployed`' (безработные), '`College_jobs`' (трудоустроенные на должности, требующие высшее образование), но при этом добавлен целевой признак '`Major_category`' (группа специальностей).

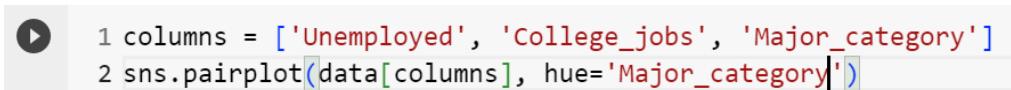


Рисунок 2.10 – Попарное распределение признаков

Результат представлен на рис. 2.11.

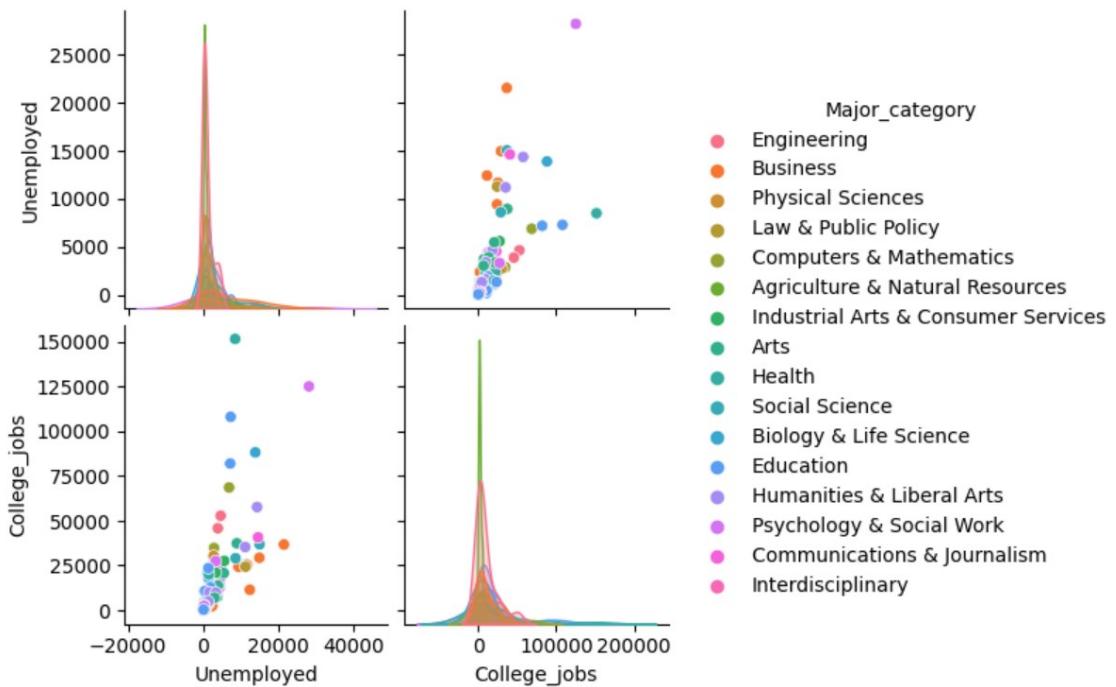


Рисунок 2.11 – Попарное распределение признаков с целевым признаком

На рисунке 2.12 показан пример использования графика **scatter** библиотеки **matplotlib**, предназначенного для вывода множества точек. Данный график также позволяет рассматривать корреляцию между двумя признаками.

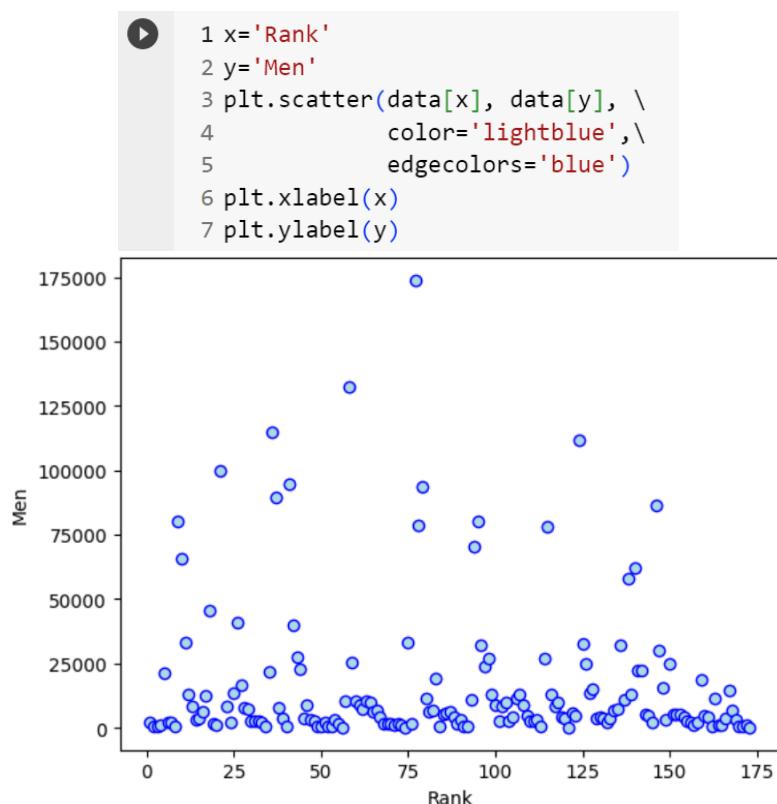


Рисунок 2.12 – График **scatter** библиотеки **matplotlib**

В реальных задачах машинного обучения при первичном анализе данных необходимо выявить корреляции признаков обучающей выборки. В пакете **pandas** имеется встроенный инструмент для этого – метод **corr( )** класса **DataFrame**. На рисунке 2.13 показан фрагмент вывода этой функции.



```

1 cols=['Employed','Median earning','Full_time','Rank', 'Men', 'Women']
2 data[cols].corr()

```

|                       | <b>Employed</b> | <b>Median earning</b> | <b>Full_time</b> | <b>Rank</b> | <b>Men</b> | <b>Women</b> |
|-----------------------|-----------------|-----------------------|------------------|-------------|------------|--------------|
| <b>Employed</b>       | 1.000000        | -0.107547             | 0.995838         | 0.070751    | 0.870605   | 0.944037     |
| <b>Median earning</b> | -0.107547       | 1.000000              | -0.082258        | -0.873308   | 0.025991   | -0.182842    |
| <b>Full_time</b>      | 0.995838        | -0.082258             | 1.000000         | 0.034725    | 0.893563   | 0.917681     |
| <b>Rank</b>           | 0.070751        | -0.873308             | 0.034725         | 1.000000    | -0.094780  | 0.174913     |
| <b>Men</b>            | 0.870605        | 0.025991              | 0.893563         | -0.094780   | 1.000000   | 0.672759     |
| <b>Women</b>          | 0.944037        | -0.182842             | 0.917681         | 0.174913    | 0.672759   | 1.000000     |

Рисунок 2.13 – Определение коррелирующих признаков набора данных

Полученная матрица имеет размер  $6 \times 6$ . Это незначительный размер (в реальных задачах машинного обучения размеры матриц корреляции имеют порядки  $10^6 - 10^{10}$  и более), но даже для матрицы рассматриваемого набора данных проанализировать корреляцию признаков вручную – трудоемкая задача. Например, можно использовать скрипты, для выделения больших коэффициентов корреляции. Но лучше использовать специальный тип графика – **heatmap** (рисунок 2.14).

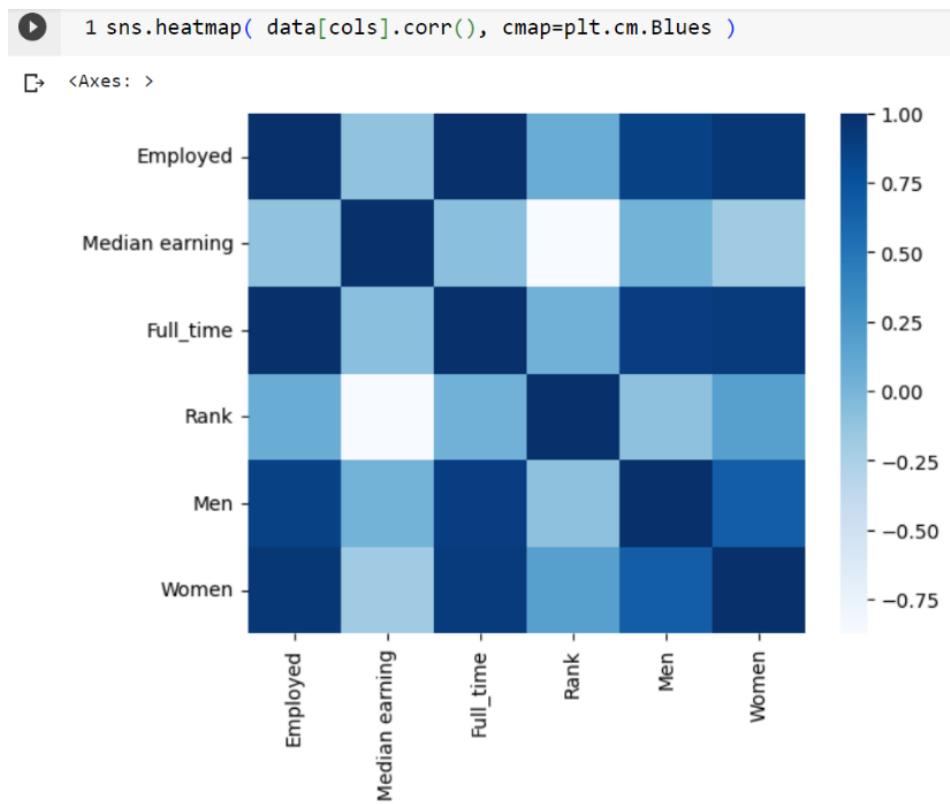


Рисунок 2.19 – Визуализация матрица корреляции с использованием графика типа **heatmap**

Коррелирующие признаки обычно удаляются и не рассматриваются в процессе обучения.

### Вопросы для самопроверки по теме 12

1. Какие библиотеки Python используются для построения графиков?
2. Какие типы графиков вы знаете?
3. Поясните назначение метода **scatter( )**.
4. Поясните назначение метода **corr( )**.
5. Поясните назначение метода **hist( )**.
6. Поясните назначение библиотеки **pandas**.
7. Поясните назначение методов **head( )** и **tail()** класса **DataFrame**.
8. Для чего предназначен тип **DataFrame** библиотеки **pandas**.
9. Для чего предназначены методы **info()** и **describe()** класса **DataFrame**.
10. Поясните назначение метода **heatmap()** библиотеки **seaborn**.

## СПИСОК ЛИТЕРАТУРЫ

1. Репозиторий с примерами кода из лекций:  
[https://github.com/enikolaev/Algorithmization\\_and\\_programming](https://github.com/enikolaev/Algorithmization_and_programming)
2. Шоу З. Легкий способ выучить Python 3 еще глубже / Зед Шоу; [перевод с английского М.А. Райтмана]. – Москва : Эксмо, 2020. – 272 с. ISBN 978-5-04-093107-1.
3. Бизли Д. Python. Исчерпывающее руководство / Бизли Дэвид; – СПб.: Питер, 2023. – 368 с. – Серия «Для профессионалов». ISBN 978-5-4461-1956-1.
4. Лучано Р. Python. К вершинам мастерства / Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2016. – 768 с.: ил. ISBN 978-5-97060-384-0.
5. Яворски М., Зиаде Т.. Лучшие практики и инструменты / Яворски Михал, Зиаде Тарек; - СПб.: Питер, 2021. – 560 с.: ил. – ISBN 978-5-4461-1589-1.
6. Седжвик Р., Уейн К., Дондеро Р. Программирование на языке Python: учебный курс. : пер. с англ. – СПб. : ООО «Альфа-книга», 2017. – 736 с. : ил. – ISBN 978-5-9908462-1-0.