Перед выполнением лабораторных работ необходимо установить последнюю версию интерпретатора Python

Как установить python - https://pythonru.com/baza-znanij/skachat-i-ustanovit-python-na-windows-10

Лабораторная работа 1: Введение в Python

Цель работы: Построение функции с использованием модуля math в языке Python.

Вопросы, изучаемые в работе:

- Основной синтаксис и структура языка Python
- Переменные, типы данных и операторы
- Операции ввода и вывода

Отступы в Python

В то время как в других языках программирования отступ в коде предназначен только для чтения, в Python отступ очень важен. Python использует отступ для указания блока кода.

```
if 5 > 2:
    print("Пять больше двух!")
```

Python заявит вам об ошибке, если вы пропустите отступ:

```
if 5 > 2:
     print("Пять больше двух!")
```

Результат:

```
  File "demo_indentation_test.py", line 2
    print("Пять больше двух!")
        ^
IndentationError: expected an indented block
```

Комментарии

Python предоставляет возможность комментирования документации внутри кода. Комментарии следует начинать с символа #, а интерпретатор отобразит остальную часть строки в виде комментария:

```
# Это комментарий.
print("Привет, Мир!")
```

Ввод, вывод данных и переменные в Python

Большинство программ, даже самых простых, выполняют обработку какой-либо информации – получают разнообразные данные, производят необходимые операции, после чего выводят результат. За ввод и вывод данных в Python отвечают встроенные функции **input()** и **print()**. С помощью функции вывода print() можно написать классическую программу Hello, World! в одну строку:

```
>>> print('Hello, World!')
Hello, World!
```

Для ввода нужной информации используют **input()**. В этом примере переменная **name** с помощью оператора присваивания = получит введенное пользователем значение:

```
name = input()
```

Чтобы пользователю было **понятнее**, какое именно значение от него ожидает программа, можно добавить пояснение:

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

Или:

```
name = input('Введите свое имя ')
```

Напишем программу, которая запрашивает имя пользователя и выводит приветствие:

```
name = input('Как тебя зовут? ')
print('Привет,', name)
```

Результат:

```
Как тебя зовут? Вася
Привет, Вася
```

В этой программе используются две встроенные функции **`input()`** и **`print()`**, а также переменная name. Переменная – это именованная область памяти, в которой во время выполнения программы хранятся данные определенного типа (о типах данных расскажем ниже).

В стандартах оформления кода PEP 8 (https://peps.python.org/pep-0008/) даны рекомендации по названиям переменных:

– Названия не должны начинаться с цифры, но могут заканчиваться цифрой. Например, назвать переменную **`7up`** – неправильно, а так – **`seven11`** – можно;

– Названия могут состоять из комбинации строчных, заглавных букв, цифр и символов подчеркивания: **`lower_case, mixedCase, CapitalizedCase, UPPER_CASE, lower123;`**

– Не следует давать переменным названия, совпадающие со служебными словами, названиями встроенных функций и методов, к примеру – **`print, list, dict, set, pass, break, raise;`**

– Следует избегать использования отдельных букв, которые могут быть ошибочно приняты друг за друга – l (**L** в нижнем регистре), I (**i** в верхнем регистре) или за нуль – O;

– В названиях не должно быть пробелов, дефисов и специальных символов, например, ' или $;

– Главный принцип именования переменных – осмысленность. По названию переменной должно быть понятно, какого рода данные в ней хранятся;

– Например, **car_model, pet_name, CARD_NUMBER** более информативны, чем **a, a1, a2.**

Переменные выполняют две важные функции:

– Делают код понятнее;

– Дают возможность многократно использовать введенные данные.

Если программа небольшая, а введенное значение используется однократно, можно обойтись без использования переменной:

```
print('Привет,', input('Как тебя зовут?'))
```

Ввод и вывод нескольких переменных, f-строки

Если в программе используются несколько переменных, ввод данных можно оформить на отдельных строках:

```
first_name = input()
last_name = input()
age = input()
```

Или в одну строку:

```
first_name, last_name, age = input(), input(), input()
```

Либо так – если значения переменных равны:

```
x1 = x2 = x3 = input()
```

Чтобы вывести значения переменных на экран, названия перечисляют в **`print()`** через запятую:

```
print(first_name, last_name, age)
```

Или по отдельности:

```
print(first_name)
print(last_name)
print(age)
```

При перечислении через запятую Python выводит все переменные в одну строку, разделяя значения пробелами:

```
Иванов Иван 12
```

Вместо пробела можно подставить любой другой разделитель. Например:

```
print(first_name, last_name, age, sep="***")
```

В результате значения будут разделены звездочками:

```
Иванов***Иван***12
```

Чтобы сделать вывод более информативным, используют f-строки:

```
print(f'Имя: {first_name}, Фамилия: {last_name}, Возраст: {age}')
```

Все содержимое такой строки находится в конструкции f'...', а названия переменных внутри строки заключаются в фигурные скобки {...}.

Операции во время вывода

Функция **print()**, помимо вывода результатов работы программы, допускает проведение разнообразных операций с данными:

```
>>> print(5 + 5)
10
>>> print(10 // 3)
3
>>> print(6 ** 2)
36
>>> print('I' + ' love' + ' Python')
I love Python
```

Встроенные типы данных в Python

Питон работает с двумя категориями данных – встроенными типами (они поддерживаются по умолчанию) и специализированными (для операций с ними нужно подключение определенного модуля). К специализированным типам данных относятся, например, **datetime** (дата и время) и **deque** (двухсторонняя очередь).

Все встроенные типы данных в Python можно разделить на следующие группы:

- **Числовые** – целые, вещественные, комплексные числа. Примечание: для максимально точных расчетов с десятичными числами в Python используют модуль **decimal** (тип данных Decimal), а для операций с рациональными числами (дробями) – модуль **fractions** (тип данных Fraction).
- **Булевы** – логические значения **True** (истина) и **False** (ложь).
- **Строковые** – последовательности символов в кодировке **Unicode**.

– **NoneType** – нейтральное пустое значение, аналогичное `null` в других языках программирования.

– **Последовательности** – списки, кортежи, диапазоны.

– **Словари** – структура данных типа «ключ: значение».

– **Множества** – контейнеры, содержащие уникальные значения. Подразделяются на изменяемые set и неизменяемые frozenset множества.

– **Байтовые** типы – `bytes` (байты), `bytearrayv` (изменяемая байтовая строка), `memoryview` (предоставление доступа к внутренним данным объекта).

В таблице 1 приведены примеры и определения встроенных типов данных:

Таблица 1. Описание встроенных типов данных.

Тип данных	Значение	Определение в Python	Вариант использования
Целые числа	-3, -2, -1, 0, 1, 2, 3	int	a = int(input())
Вещественные числа	-1.5, -1.1, 0.6, 1.7	float	a = float(input())
Комплексные числа	−5i, 3+2i	complex	a = complex(input())
Булевы значения	True, False	True, False	flag = True
NoneType	None	None	a = None

Строка	'abracadabra'	str	a = str(5)
Список	[1, 2, 3], ['a', 'b', 'c']	list	a = list(('a', 'b', 'c'))
Кортеж	('red', 'blue', 'green')	tuple	a = tuple(('red', 'blue', 'green'))
Изменяемое множество	{'black', 'blue', 'white'}, {1, 3, 9, 7}	set	a = set(('black', 'blue', 'white'))
Неизменяемое множество	{'red', 'blue', 'green'}, {2, 3, 9, 5}	frozenset	a = frozenset((2, 5, 3, 9))
Диапазон	0, 1, 2, 3, 4, 5	range	a = range(6)
Словарь	{'color': 'red', 'model': 'VC6', 'dimensions': '30x50'}	dict	a = dict(color='red', model='VC6', dimensions='30x50')
Байты	b'\x00\x00\x00'	bytes	a = bytes(3)
Байтовая строка	(b'\x00\x00')	bytearray	a = bytearray(2)

Просмотр памяти	0x1477a5813a00	memoryview	a = memoryview(bytes(15))

Чтобы узнать тип данных, нужно воспользоваться встроенной функцией **`type()`:**

```
>>> a = 3.5
>>> type(a)
<class 'float'>
```

Как задать тип переменной

Важно заметить, что если тип переменной не указан **явно** при вводе, т.е. ввод выполняется как **`a = input()`**, то Python будет считать введенное значение **строкой**. В приведенном ниже примере Питон вместо сложения двух чисел выполняет конкатенацию строк:

```
>>> a, b = input(), input()
5
6
>>> print(a + b)
56
```

Это произошло потому, что **`a`** и **`b`** были введены как строки, а не целые числа:

```
>>> type(a)
<class 'str'>
>>> type(b)
```

```
<class 'str'>
```

Чтобы ввести целое число, следует использовать конструкцию **int(input()),** вещественное –**float(input()).**

Математические операции в Python

Все операции в математике имеют определенный приоритет: сначала выполняется возведение в степень, затем деление по модулю и так далее. Этот приоритет соблюдается и в Питоне (смотри таблицу 2):

Таблица 2. Описание математических операций.

Приоритет	**Оператор Python**	**Операция**	**Пример**	**Результат**
1	**	Возведение в степень	5 ** 5	3125
2	**%**	Деление по модулю (получение остатка)	16 % 7	2
3	//	Целочисленное деление (дробная часть отбрасывается)	13 // 3	4
4	/	Деление	39 / 2	19.5

5	*	Умножение	123 * 321	39483
6	-	Вычитание	999 – 135	864
7	+	Сложение	478 + 32	510

Python допускает применение сложения и умножения в операциях со строками. Сложение строк, как уже упоминалось выше, называется **конкатенацией**:

```
>>> print('Python -' + ' лучший' + ' язык' + ' программирования')
Python - лучший язык программирования
```

Умножение строки на целое число называется **репликацией**:

```
>>> print('Репликанты' * 5)
РепликантыРепликантыРепликантыРепликантыРепликанты
```

Однако попытки умножить строки друг на друга или на вещественное число обречены на провал:

```
>>> print('Репликанты' * 5.5)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'float'
```

Операторы присваивания

Операторы присваивания используются для присваивания значений переменным (смотри таблицу 3):

Таблица 3. Описание операторов присваивания.

Оператор	**Пример**	**Так же как**
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x — 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3

Оператор	**Пример**	**Так же как**
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Операторы сравнения

Операторы сравнения используются для сравнения двух значений (смотри таблицу 4):

Таблица 4. Описание операторов сравнения.

Оператор	**Значение**	**Пример**
==	равно	x == y
!=	не равно	x != y
>	больше чем	x > y
<	меньше чем	x < y

Оператор	Значение	Пример
>=	больше чем или равно	x >= y
<=	меньше чем или равно	x <= y

Логические операторы

Логические операторы используются для объединения условных операторов (смотри таблицу 5):

Таблица 5. Описание логических операторов.

Оператор	Значение	Пример
and	Возвращает значение True если оба утверждения верны	x < 5 and x < 10
or	Возвращает True если одно из утверждений верно	x < 5 or x < 4
not	Меняет результат, возвращает False если результат True	not(x < 5 and x < 10)

Операторы тождественности в Python

Операторы тождественности используются для сравнения объектов. Являются ли они одним и тем же объектом с одинаковым местоположением в памяти (смотри таблицу 6):

Таблица 6. Описание операторов тождественности.

Оператор	**Значение**	**Пример**
is	Возвращает true если переменные являются одним объектом	x is y
is not	Возвращает true если переменные разные	x is not y

Операторы принадлежности

Операторы принадлежности используются для проверки того, представлена ли последовательность в объекте (смотри таблицу 7):

Таблица 7. Описание операторов принадлежности.

Оператор	Значение	Пример
in	Возвращает True если последовательность присутствует в объекте	x in y
not in	Возвращает True если последовательность не присутствует в объекте	x not in y

Побитовые операторы

Побитовые операторы используются для работы в битовом (двоичном) формате (смотри таблицу 7):

Таблица 7. Описание побитовых операторов.

Оператор	Название	Значение
&	И	Устанавливает каждый бит в 1, если оба бита 1
|	Или	Устанавливает каждый бит в 1 если один из двух битов 1
^	только или	Устанавливает каждый бит в 1 если только один из битов 1
~	Не	Переставляет все биты

Оператор	Название	Значение
<<	Сдвиг влево	Сдвигает влево на количество бит указанных справа
>>	Сдвиг вправо	Сдвигает вправо на количество бит указанных справа

Преобразование типов данных

Python позволяет на лету изменять типы данных. Это может оказаться очень полезным при решении тренировочных и практических задач.

Округление вещественного числа:

```
>>> a = float(input())
5.123
>>> print(int(a))
5
```

Преобразование целого числа в вещественное:

```
        >>> a = 5
>>> print(float(a))
5.0
```

Преобразование строки в число и вывод числа без ведущих нулей:

```
>>> a = '00032567'
>>> print(int(a))
32567
```

Модуль Math

Встроенный модуль **math** в Python предоставляет набор функций для выполнения математических, тригонометрических и логарифмических операций. Некоторые из основных функций модуля:

- **pow(num, power)**: возведение числа num в степень power
- **sqrt(num):** квадратный корень числа num
- **ceil(num):** округление числа до ближайшего наибольшего целого
- **floor(num):** округление числа до ближайшего наименьшего целого
- **factorial(num):** факториал числа
- **degrees(rad):** перевод из радиан в градусы
- **radians(grad):** перевод из градусов в радианы
- **cos(rad):** косинус угла в радианах
- **sin(rad):** синус угла в радианах
- **tan(rad):** тангенс угла в радианах
- **acos(rad):** арккосинус угла в радианах
- **asin(rad):** арксинус угла в радианах
- **atan(rad):** арктангенс угла в радианах
- **log(n, base):** логарифм числа n по основанию base
- **log10(n):** десятичный логарифм числа n

Варианты заданий:

№ вар.	Программируемая формула	*A*	*B*	*C*	*D*	Результат
1	$\sqrt{\frac{A}{2\pi\left(B-\sqrt{B^2-C^2}\right)}}+\sin D$	10^5	5	2	2.5	1.95862E+2
2	$\sqrt{A/(2D*\ln\frac{C}{B})}-\frac{3.5}{C}+\ln\lvert D\rvert$	10^4	10	0.1	-3	-1.48774E+1
3	$\sqrt{\frac{A}{\left(2\pi D\left(\ln\frac{88D}{B}-1.75\right)\right)}}-3.5C$	10^4	10	0.2	3	1.79615E+1
4	$\sqrt{A\cdot(3D+9B+10C)/(25\pi D^2)}$	10^{-2}	-1.5	4.1	-3	1.61778E-2
5	$\sqrt{\frac{\ln A\cdot(B+C)}{4D\cdot A(B-C)}}-\exp\left(\frac{\sin B}{\cos C}\right)$	10^1	-1.7	3.9	-3	-3.83304E+0
6	$\left(\frac{A}{B(C+D)\ln\left(A\frac{C+D}{C-D}\right)}\right)^{3/5}$	10^3	3.5	4.1	-3	1.06442E+1
7	$\sqrt{A\cdot B\frac{\left(1+C\cdot\exp\left(D/(A\cdot B)\right)\right)}{4\pi D}}$	10^1	-0.5	1.1	-1	9.65643E-1
8	$1.25\left(\frac{5.98A}{\exp\frac{B(C+1.41)^{0.5}}{87}}-D\right)$	10^2	-20.5	5.1	-1.5	1.36556E+3
9	$0.25\left(\frac{4D+A}{\exp\frac{B\sqrt{C}}{60}}-2.1D\right)$	10^{-1}	2.5	5.1	-1.5	-5.55037E-1
10	$0.25\left(\frac{\sin A}{5.2\text{le}{-6}\cdot\exp B}-4\sqrt{C\ln C}\right)-\pi D^2$	10^{-1}	1.2	5.1	2.05	1.42678E+3

№ вар.	Программируемая формула	*A*	*B*	*C*	*D*	Результат
11	$6.28-\frac{\sin(\ln A)+\cos(\lg B)}{34.2*10^{-3}\exp(\sqrt[4]{C})}$	10^3	12	7.21	-	2.79759E-1
12	$\sqrt{\frac{10\,A+2\pi\cdot D}{1.1+\sqrt{\frac{13\,B}{0.5+\sqrt{\cos\,C}}}}}$	10	1.3	0.1	-.05	4.72802E+0
13	$7.7\cdot10^{-5}\cdot\sqrt{5.5^{A}-\frac{2\sin(A+B)}{1-\cos(\ln C)}+\frac{\pi}{D}}$	10^{-2}	1.39	3.1	0.55	1.39860E-4
14	$12A+\sqrt{7.41\left(B+\sin\left(\frac{C}{4}\right)\right)}-0.803\left(B\cdot\cos\frac{D}{3}+\sqrt{A}\right)$	10^{-3}	21.39	23.1	-0.12	-4.73017E+0
15	$\frac{A^{B}+B^{A}\ln C-C\cdot\lg\sqrt{A}}{2B+D}$	10^{-1}	2.1	0.1	-3.12	-2.24257E+0
16	$\frac{4\pi}{3}A^{3}-\frac{2.1B\cdot10^{C+1}}{C+1-D\cdot\exp A}+\sqrt{C+1}$	10^{-3}	-2.1	1.1	-3.12	1.07743E+2
17	$\sqrt{B^{1/3}+2.4A^{3/13}}+\sin\left(A^{3/13}\cdot\frac{C-D}{2\pi}\right)$	10^4	122.2	1.1	-3.12	4.39587E+0
18	$\frac{A}{B}\sqrt{\frac{(C-1)^2}{5.4B}+\frac{0.015(C-1)}{5.4A}-1+C}$	10^3	33.3	2.1	-	3.15920E+1
19	$\sqrt{\frac{A\ln D}{2-\pi+\ln D\cdot(\cos(\ln D-31*10^{-3}))}}$	-10^3	-	-	10	2.96095E+1
20	$\sqrt{A\cdot(3\sin D-9\cos\lvert B\rvert+10\,\mathrm{tg}C)}\Big/25\pi\cdot D^{2}$	-10^4	0.2	-0.5	3	5.26688E-1
21	$\sqrt{\frac{A((B+\cos C))-0.3C}{4D\cdot B\cdot\exp(B+\cos C)}}$	10^4	7.7	-0.9	0.77	9.38646E-1
22	$\left(\frac{\pi A(C-D)}{B(C+D)\cdot\ln\left(B\cdot\frac{C+D}{C-D}\right)}\right)^{3/8}\cdot\sin(C-D)$	10^3	-0.88	0.9	1.77	-1.08136E+1
23	$10^{-7}\left(\frac{5.98\cdot\exp(C+1.4)}{0.5+B}-\ln(C+1.4)\right)\cdot\cos\frac{A(C+1.4)}{1.81}$	10^{-1}	-0.33	2.2	-	1.28586E-4

№ вар.	Программируемая формула	*A*	*B*	*C*	*D*	Результат
24	$1.5\left(\frac{A/3}{\sin\frac{B\sqrt{\lvert C\rvert}}{A/3}}-\frac{3D}{A}\right)$	10^2	-0.33	-3.3	10	-2.78081E+3
25	$10^6\frac{tgA}{3.1\cdot10^6}-\sqrt[3]{\lvert C\ln\lvert C\rvert\rvert}-\pi\cdot B\cdot D^3$	10^{-1}	-0.83	-4.4	1.4	5.31933E+0
26	$1.2\cdot10^{-3}+\frac{\cos\left(\ln\left(A/B\right)\right)+\cos\left(\lg\left(A/B\right)\right)}{4.2\cdot10^{-3}\exp\left(\sqrt[4]{\lvert\ln\left(A/B\right)\rvert}\right)}$	10^3	5	-	-	-1.40486E+1
27	$-11\cdot10^{-5}\sqrt{5B^A+\frac{2\sin(A+B)}{1-\cos(\ln C)}-\frac{\pi}{(A+B)B^A}}$	2.5	10	0.5	-	-4.37319E-3
28	$2\cdot10^A+\sqrt{4.7\lvert D+\sin\frac{C}{2}\rvert+0.803B\cdot\exp\frac{D}{3}}$	1.9	10^3	-2.1	13.5	4.27833E+2
29	$\frac{A^5+B^5\ln5-C\cdot\lg\sqrt{5B}}{5B+D/500}$	1.09	10^{-2}	-2.4	10^3	-1.10303E-2
30	$\frac{2\sqrt{B+1}}{3\cdot\lg B}\cdot\frac{\pi}{C}\cdot A^3-\frac{10^{B+1}}{\exp(B+1)}+\sqrt{B+1}$	10.3	0.2	-10^4	-	-3.31949E+0
31	$\sqrt{C^{D/3}-0.5A^{3/2}+\exp\left(A^{3/2}\cdot\frac{C+D}{2A}\right)}$	10^{-2}	-	10^2	-2.5	1.14453E+1

Лабораторная работа 2: Управляющие структуры

Цель работы: Построение программы с использованием условных конструкций и циклических структур в языке Python.

Вопросы, изучаемые в работе:

- Условные операторы (if, else, elif)
- Циклические структуры (while, for)
- Использование управляющих структур для решения задач

Условные конструкции используют условные выражения и в зависимости от их значения направляют выполнение программы по одному из путей. Одна из таких конструкций - это конструкция **if**. Она имеет следующее формальное определение:

```
if логическое_выражение:
    инструкции
[elif логическое выражение:
    инструкции]
[else:
    инструкции]
```

В самом простом виде после ключевого слова **if** идет логическое выражение. И если это логическое выражение возвращает **True**, то выполняется последующий блок инструкций, каждая из которых должна начинаться с новой строки и должна иметь отступы от начала выражения **if** (отступ желательно делать в 4 пробела или то количество пробелов, которое кратно 4):

```
language = "english"
if language == "english":
    print("Hello")
print("End")
```

Поскольку в данном случае значение переменной **language** равно **"english"**, то будет выполняться блок **if**, который содержит только одну инструкцию - **print("Hello")**. В итоге консоль выведет следующие строки:

```
Hello
End
```

Обратите внимание в коде на последнюю строку, которая выводит сообщение **"End"**. Она не имеет отступов от начала строки, поэтому она не

принадлежит к блоку **if** и будет выполняться в любом случае, даже если выражение в конструкции **if** возвратит **False**.

Но если бы мы поставили бы отступы, то она также принадлежала бы к конструкции **if:**

```
language = "english"

if language == "english":
    print("Hello")
    print("End")
```

Блок else

Если вдруг нам надо определить альтернативное решение на тот случай, если выражение в **if** возвратит **False**, то мы можем использовать блок **else**:

```
language = "russian"
if language == "english":
    print("Hello")
else:
    print("Привет")
print("End")
```

Если выражение **language == "english"** возвращает **True**, то выполняется блок **if**, иначе выполняется блок **else**. И поскольку в данном случае условие **language == "english"** возвращает **False**, то будут выполняться инструкция из блока **else**.

Причем инструкции блока else также должны имет отступы от начала строки. Например, в примере выше **print("End")** не имеет отступа, поэтому она не входит в блок **else** и будет выполняться вне зависимости, чему равно условие **language == "english".** То есть консоль нам выведет следующие строки:

```
Привет
End
```

Блок **else** также может иметь несколько инструкций, которые должны иметь отступ от начала строки:

```
language = "russian"
if language == "english":
    print("Hello")
    print("World")
else:
    print("Привет")
    print("мир")
```

elif

Если необходимо ввести несколько альтернативных условий, то можно использовать дополнительные блоки **elif**, после которого идет блок инструкций.

```
language = "german"
if language == "english":
    print("Hello")
    print("World")
elif language == "german":
    print("Hallo")
    print("Welt")
else:
    print("Привет")
    print("мир")
```

Сначала Python проверяет выражение **if**. Если оно равно **True**, то выполняются инструкции из блока **if**. Если это условие возвращает **False**, то Python проверяет выражение из **elif**.

Если выражение после **elif** равно **True**, то выполняются инструкции из блока **elif**. Но если оно равно **False** то выполняются инструкции из блока **else.**

При необходимости можно определить несколько блоков **elif** для разных условий. Например:

```
language = "german"
if language == "english":
    print("Hello")
elif language == "german":
    print("Hallo")
elif language == "french":
    print("Salut")
else:
    print("Привет")
```

Вложенные конструкции if

Конструкция if в свою очередь сама может иметь вложенные конструкции **if**:

```
language = "english"
daytime == "morning"
if language == "english":
    print("English")
    if daytime == "morning":
        print("Good morning")
    else:
        print("Good evening")
```

Здесь конструкция if содержит вложенную конструкцию **if/else**. То есть если переменная **language** равна **"english",** тогда вложенная конструкция **if/else** дополнительно проверяет значение переменной **daytime** - равна ли она

строке **"morning"** ли нет. И в данном случае мы получим следующий консольный вывод:

```
English
Good morning
```

Стоит учитывать, что вложенные выражения **if** также должны начинаться с отступов, а инструкции во вложенных конструкциях также должны иметь отступы. Отступы, расставленные не должным образом, могут изменить логику программы. Так, предыдущий пример НЕ аналогичен следующему:

```
language = "english"
daytime = "morning"

if language == "english":
    print("English")
if daytime == "morning":
    print("Good morning")
else:
    print("Good evening")
```

Подобным образом можно размещать вложенные конструкции **if/elif/else** в блоках **elif** и **else**:

```
language = "russian"
daytime = "morning"
if language == "english":
    if daytime == "morning":
        print("Good morning")
    else:
        print("Good evening")
else:
    if daytime == "morning":
```

```
        print("Доброе утро")
    else:
        print("Добрый вечер")
```

Циклы

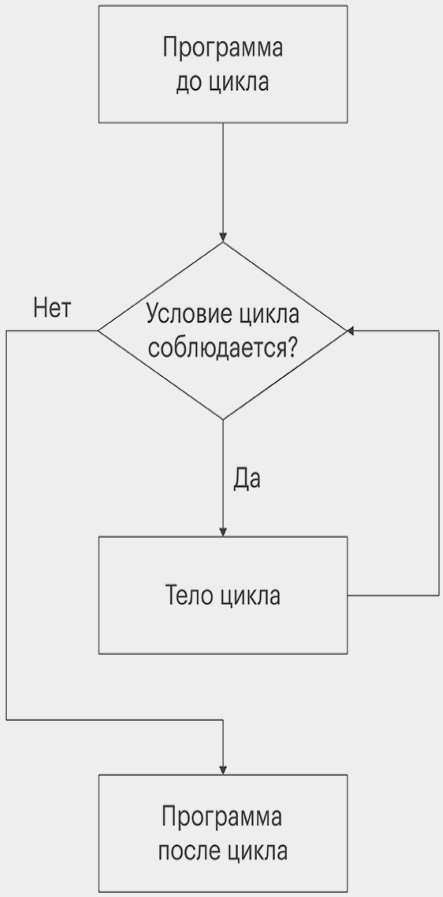
Любой цикл состоит из двух обязательных элементов:

Условие — начальный параметр; цикл запустится только при его выполнении и закончится, как только условие перестанет выполняться;

Тело — сама программа, которая выполняется внутри цикла.

Схематически его можно представить так:

Схема цикла



В синтаксисе Python в конце строки с условием ставится двоеточие, а всё тело выделяется отступом (табуляцией или четырьмя пробелами).

```
программа до цикла
условие:
    первая строка тела
    вторая строка тела
программа после цикла
```

Цикл while в Python

While — наиболее простой и понятный вид цикла. Его ещё называют циклом с предусловием.

```
x = 1
while x < 5:
    print(x)
    x += 1  # Означает то же самое, что x = x + 1


>>> 1
>>> 2
>>> 3
>>> 4
```

С языка Python на русский программу можно перевести так: «Пока икс меньше пяти, печатай икс и прибавляй к нему единицу».

Но в простоте **while** кроется и опасность: его легко сделать бесконечным. Например, если в коде выше мы уберём **x += 1**, то получится вот так:

```
# Этот код будет выполняться бесконечно
x = 1
while x < 5:
    print(x)
```

Здесь с переменной x ничего не происходит. Она всегда равна единице, поэтому условие цикла никогда не перестанет выполняться. Соответственно, он никогда не завершится.

Чтобы избежать таких ситуаций, при использовании **while** нужно следить: перестанет ли когда-нибудь выполняться условие? Ещё можно использовать оператор **break** — о нём мы расскажем чуть позже.

Цикл for в Python

Цикл for программисты используют куда чаще, чем **while**. Для него мы устанавливаем не условие в чистом виде, а некий массив данных: список, кортеж, строку, словарь, диапазон или любой другой итерируемый объект.

На каждой итерации цикла программа как бы спрашивает: «Остались ли в объекте ещё элементы, по которым я не прошла?»

Допустим, у нас есть список с числами: **[14, 101, -7, 0]**. Мы можем использовать его вместе с **for**, чтобы напечатать каждый элемент по отдельности.

```
num_list = [14, 101, -7, 0]
for number in num_list:
    print(number)

>>> 14
>>> 101
>>> -7
>>> 0
```

Здесь переменная number обновляется при каждом новом витке цикла. Сначала она хранит в себе первый элемент, потом второй, и так — пока список не закончится.

Как и любую другую переменную, мы могли назвать **number** как угодно. Часто используют буквы **i**, **j** и **k**. Если внутри цикла мы ни разу не обращаемся

к этой переменной, то среди питонистов её принято обозначать символом нижнего подчёркивания _.

Функция range()

Когда нужно применить for к числовому промежутку, его можно **задать диапазоном**. Для этого используют функцию `range()`. В неё можно передать от одного до трёх аргументов.

Если аргумент один, то сформируется диапазон от нуля до числа, предшествующего значению аргумента.

```
for i in range(3):
    print(i)

>>> 0
>>> 1
>>> 2
```

Если аргумента два, то сформируется диапазон от значения первого аргумента до числа, предшествующего значению второго аргумента.

```
for i in range(23, 26):
    print(i)
>>> 23
>>> 24
>>> 25
```

Если аргумента три, то первые два работают, как в прошлом случае. Третий же означает шаг, с которым числа следуют друг за другом.

```
for i in range(10, 20, 3):
    print(i)

>>> 10
>>> 13
```

```
>>> 16
>>> 19
```

Прерывание цикла: ключевое слово break

Бывают случаи, когда нужно завершить цикл принудительно, даже если его условие всё ещё выполняется. В таких случаях используют ключевое слово **`break`**.

Возьмём строку **`Hi, loop!`** и будем выводить каждый её символ. Если встретим запятую, досрочно завершим цикл.

```
string = 'Hi, loop!'
for i in string:
    if i == ',':
        break
    print(i)

>>> H
>>> i
```

Если же в строке запятой не будет, то цикл пройдёт по каждому её символу — и только потом завершится.

```
string = 'Hi loop!'
for i in string:
    if i == ',':
        break
    print(i)

>>> H
>>> i
>>>
```

```
>>> l
>>> o
>>> o
>>> p
>>> !
```

Пропуск части цикла: ключевое слово continue

Иногда возникает необходимость принудительно начать следующий шаг цикла, пропустив часть строк в его теле. Для таких случаев существует ключевое слово continue.

Возьмём числа от 1 до 10 включительно и выведем из них только те, которые не делятся ни на 2, ни на 3.

```
for i in range(1, 10):
    if i % 2 == 0 or i % 3 == 0:
        continue
    print(i)
>>> 1
>>> 5
>>> 7
```

Как видим, если срабатывает условие **if** (то есть если число делится на 2 или на 3 без остатка), то оставшаяся часть тела не работает — и **i** не печатается.

Последнее действие в цикле: ключевое слово else

Обычно ключевое слово **else** употребляют в связке с **if**, но у него есть и другое применение. Его можно использовать вместе с **while** или **for**. В таком случае **else**-код выполнится после того, как пройдут все витки цикла.

Если же цикл досрочно прервётся из-за **break**, то часть программы в else не выполнится.

Вспомним наш код со строкой **Hi, loop!** и добавим к нему **else**.

```
string = 'Hi, loop!'
for i in string:
    if i == ',':
        break
    print(i)
else:
    print('Цикл завершился без break')

>>> H
>>> i
```

В строке была запятая, сработал **break** — не выполнилось **else**-условие. Теперь уберём из неё запятую и посмотрим, что получится.

```
string = 'Hi loop!'
for i in string:
    if i == ',':
        break
    print(i)
else:
    print('Цикл завершился без break')

>>> H
>>> i
```

```
>>>
>>> l
>>> o
>>> o
>>> p
>>> !
>>> Цикл завершился без break
```

Цикл прошёл все свои итерации, завершился самостоятельно, и поэтому код в **else** выполнился. Он также будет работать, если цикл не совершил ни одного витка.

```
while 1 == 0:
    print('Эта строка никогда не выполнится')
else:
    print('Цикл завершился без break')
>>> Цикл завершился без break
```

Бесконечный цикл

Иногда использовать бесконечный цикл может быть хорошей идеей. Например, мы пишем игру: она должна работать до тех пор, пока игрок из неё не выйдет. В этом случае в условии выхода нужно будет прописать break.

Чтобы цикл был бесконечным, его **условие должно выполняться всегда**. Это можно сделать разными способами.

```
# Способ №1 – «пока истинно»
while True:
    pass  # pass – оператор-заглушка, который ничего не делает
```

Если сделать **while** False, то цикл, наоборот, никогда не начнётся.

```
# Способ №2 – «пока проверяемое значение – любое ненулевое число»
```

```
while 1:
    pass
while -4:
    pass
while 2023:
    pass
```

Если сделать **`while 0`**, то цикл никогда не начнётся.

```
# Способ №3 – «пока проверяемое значение – непустой элемент»
while 'string':
    pass
while [False, 'list', 0]:
    pass
```

Если после **`while`** поставить пустой элемент — **например**, строку **`str()`** или список **`list(),`** то цикл никогда не начнётся.

```
# Способ №4 – корректное уравнение
while 1 == 1:
    pass
while 0 != 1:
    pass
```

Резюмируем

- **Циклы** — один из основных инструментов любого Python-разработчика. С их помощью всего за пару строчек кода можно совершить сразу **множество повторяющихся действий**.

- Циклы состоят из **условия** и **тела**. Код в теле выполняется только до тех пор, пока соблюдено условие.
- В Python есть **два вида циклов**: **`while`** и **`for`**. В **`while`** условие задаётся явным образом. В for перебирается каждый элемент коллекции.
- К обоим видам можно применять разные **операторы**: **`break`** для прерывания, **`continue`** для пропуска части тела, **`else`** для совершения последнего действия перед выходом из цикла.
- Циклы можно **делать бесконечными** (тогда программа никогда не завершится или завершится только при выполнении определённого условия) и **вкладывать друг в друга**.

Требования к выполнению лабораторной работы:

- Разрешены к использованию следующие функции: **`print`**, **`input`**, **`range`**.
- Массивы использовать **запрещено**.

– Необходимо реализовать **проверку** на ввод чисел

В примере 1 представлена программа, которая вводит 10 целых чисел и выводит их среднее арифметическое.

Пример 1: нахождение среднего арифметического 10 введенных чисел

```
a, i = int(0), int(0)
total, mean = float(0), float(0)
for i in range(10):
    a = int(input("Input number: "))
    total = total + a
mean = total / 10
print("Mean value: ", mean)
```

Варианты заданий:

1. Перевести введенное целое число в двоичную систему и вывести его.

2. Ввести N натуральных чисел. Завершить ввод 0-м. Вывести максимальное число.

3. Ввести N натуральных чисел. Завершить ввод 0-м. Вывести номер максимального числа.

4. Ввести целое число. Вывести TRUE, если число является степенью числа 3, и FALSE в противном случае.

5. Ввести действительное число x, вычислить и вывести y = x^10 + 2 * x^9 + 3 * x^8 + ... + 10 * x + 11.

6. Ввести действительное число x, вычислить и вывести y = 11 * x^10 + 10 * x^9 + 9 * x^8 + ... + 2 * x + 1.

7. Ввести 10 действительных чисел, вывести максимальное по абсолютной величине число.

8. Ввести целые числа N и K. Вычислить и вывести число сочетаний из N по K.

9. Вычислить и вывести номер первого элемента последовательности Фибоначчи, которое превышает 1000.

10. Ввести целое число N. Вывести сумму первых N членов последовательности Фибоначчи.

11. Ввести 10 вещественных чисел. Вывести разность между максимальным и минимальным из них.

12. Ввести целое число N. Вывести количество десятичных цифр, необходимых для представления этого числа.

13. Ввести целое число N. Вывести все простые числа из диапазона [2, N].

14. Ввести целые числа N и M. Вывести TRUE, если M делит N или FALSE, если это не так. Операцию деления не использовать (%, /, // запрещены).

15. Ввести целое число N. Вывести все простые делители этого числа.

16. Ввести 10 положительных действительных чисел, вывести число с наименьшей дробной частью.

17. Ввести целое число N, вывести его в 8-ричной системе счисления.

18. Ввести 10 целых чисел, вывести минимальную по абсолютной величине разность между соседними числами.

19. Ввести целое число N, вывести его в 3-ичной системе счисления.

20. Ввести целое число N, вывести ближайшую к N степень числа 2.

21. Ввести целое число N, вывести его в 5-ичной системе счисления.

22. Ввести N натуральных чисел. Завершить ввод 0-м. Вывести номер минимального числа.

23. Ввести целое число. Вывести TRUE, если число является степенью 4 и FALSE в противном случае.

24. Ввести N чисел. Завершить ввод 0-м. Вывести номера чисел, соседи которых в сумме дают данное число.

25. Ввести целое число N, вывести его в 7-ичной системе счисления.

Лабораторная работа 3: Структуры данных – списки, кортежи, строки

Цель работы: Изучить списки, кортежи и строки. Научиться применять данные структуры при реализации задания.

Вопросы, изучаемые в работе:

– Введение в строковый тип данных в Python
– Введение в списки, кортежи в Python
– Методы и операции со списками, строками
– Работа со списками, строками для решения задач

Строки

Строковый тип **str** в Python используют для работы с любыми текстовыми данными. Python автоматически определяет тип str по кавычкам – одинарным или двойным:

```
example_string = 'Python'
type(example_string)
>>> <class 'str'>
example_string2 = "code"
type(example_string2)
>>> <class 'str'>
```

Для решения многих задач строковую переменную нужно объявить заранее, до начала исполнения основной части программы. Создать пустую переменную **str** просто:

```
example_string = ''
```

Или:

```
example_string2 = ""
```

Если в самой строке нужно использовать кавычки – например, для названия книги – то один вид кавычек используют для строки, второй – для выделения названия:

```
print("'Самоучитель Python' - возможно, лучший справочник по Питону.")
>>> 'Самоучитель Python' - возможно, лучший справочник по Питону.
print('"Самоучитель Python" - возможно, лучший справочник по Питону.')
>>> "Самоучитель Python" - возможно, лучший справочник по Питону.
```

Использование одного и того же вида кавычек внутри и снаружи строки вызовет ошибку:

```
print(""Самоучитель Python" - возможно, лучший справочник по Питону.")
  File "<pyshell>", line 1
    print(""Самоучитель Python" - возможно, лучший справочник по Питону.")
                        ^
SyntaxError: invalid syntax
```

Кроме двойных " и одинарных кавычек ', в Python используются и тройные ''' – в них заключают текст, состоящий из нескольких строк, или программный код:

```
print('''В тройные кавычки заключают многострочный текст.
Программный код также можно выделить тройными кавычками.''')
>>> В тройные кавычки заключают многострочный текст.
Программный код также можно выделить тройными кавычками.
```

Подстроки

Подстрокой называется фрагмент определенной строки. Например, 'abra' является подстрокой 'abrakadabra'. Чтобы определить, входит ли какая-то определенная подстрока в строку, используют оператор **in**:

```
example_string = 'abrakadabra'
print('abra' in example_string)
>>> True
print('zebra' in example_string)
>>> False
```

Срезы строк в Python

Индексы позволяют работать с отдельными элементами строк. Для работы с подстроками используют срезы, в которых задается нужный диапазон:

```
example_string = 'программирование'
print(example_string[7:10])
>>> мир
```

Диапазон среза **[a:b]** начинается с первого указанного элемента a включительно, и заканчивается на последнем, не включая b в результат:

```
example_string = 'программа'
print(example_string[3:8])
>>> грамм
```

Если не указать первый элемент диапазона **[:b]**, срез будет выполнен с начала строки до позиции второго элемента b:

```
example_string = 'программа'
print(example_string[:4])
>>> прог
```

В случае отсутствия второго элемента **[a:]** срез будет сделан с позиции первого символа и до конца строки:

```
example_string = 'программа'
print(example_string[3:])
>>> грамма
```

Если не указана ни стартовая, ни финальная позиция среза, он будет равен исходной строке:

```
example_string = 'позиции не заданы'
print(example_string[:])
>>> позиции не заданы
```

Шаг среза

Помимо диапазона, можно задавать шаг среза. В приведенном ниже примере выбирается символ из стартовой позиции среза, а затем каждая 3-я буква из диапазона:

```
example_string = 'Python лучше всего подходит для новичков.'
print(example_string[1:15:3])
>>> уолшв
```

Шаг может быть отрицательным – в этом случае символы будут выбираться, начиная с конца строки:

```
example_string = 'это пример отрицательного шага'
print(example_string[-1:-15:-4])
>>> а нт
```

Срез[::-1] может оказаться очень полезным при решении задач, связанных с палиндромами:

```
example_string = 'А роза упала на лапу Азора'
print(example_string[::-1])
>>> арозА упал ан алапу азор А
```

Полезные методы строк

Python предоставляет множество методов для работы с текстовыми данными. Все методы можно сгруппировать в четыре категории:

- Преобразование строк.
- Оценка и классификация строк.
- Конвертация регистра.
- Поиск, подсчет и замена символов.

Рассмотрим эти методы подробнее.

Преобразование строк

Три самых используемых метода из этой группы – **`join()`**, **`split()`** и **`partition()`**. Метод **`join()`** незаменим, если нужно преобразовать список или кортеж в строку:

```
spisok = ['Я', 'изучаю', 'Python']
example_string = ' '.join(spisok)
print(example_string)
>>> Я изучаю Python
```

При объединении списка или кортежа в строку можно использовать любые разделители:

```
kort = ('Я', 'изучаю', 'Django')
example_string = '***'.join(kort)
```

```
print(example_string)
>>> Я***изучаю***Django
```

Метод **split()** используется для обратной манипуляции – преобразования строки в список:

```
text = 'это пример текста для преобразования в список'
spisok = text.split()
print(spisok)
>>> ['это', 'пример', 'текста', 'для', 'преобразования', 'в', 'список']
```

По умолчанию **split()** разбивает строку по пробелам. Но можно указать любой другой символ – и на практике это часто требуется:

```
text = 'цвет: синий; вес: 1 кг; размер: 30x30x50; материал: картон'
spisok = text.split(';')
print(spisok)
>>> ['цвет: синий', ' вес: 1 кг', ' размер: 30x30x50', ' материал: картон']
```

Динамическая типизация

Python является языком с динамической типизацией. А это значит, что переменная не привязана жестко с определенному типу.

Тип переменной определяется исходя из значения, которое ей присвоено. Так, при присвоении строки в двойных или одинарных кавычках переменная имеет тип **str**. При присвоении целого числа Python автоматически определяет тип переменной как **int**. Чтобы определить переменную как объект **float**, ей присваивается дробное число, в котором разделителем целой и дробной части является точка.

При этом в процессе работы программы мы можем изменить тип переменной, присвоив ей значение другого типа:

```
userId = "abc"  # тип str
print(userId)
userId = 234  # тип int
print(userId)
```

С помощью встроенной функции **type()** динамически можно узнать текущий тип переменной:

```
userId = "abc"      # тип str
print(type(userId)) # <class 'str'>
userId = 234        # тип int
print(type(userId)) # <class 'int'>
```

Списки

Для работы с наборами данных Python предоставляет такие встроенные типы как списки, кортежи и словари.

Список (**list**) представляет тип данных, который хранит набор или последовательность элементов. Во многих языках программирования есть аналогичная структура данных, которая называется массив.

Создание списка

Для создания списка применяются квадратные скобки [], внутри которых через запятую перечисляются элементы списка. Например, определим список чисел:

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

Подобным образом можно определять списки с данными других типов, например, определим список строк:

```
people = ["Tom", "Sam", "Bob"]
```

Также для создания списка можно использовать функцию-конструктор **list():**

```
numbers1 = []
numbers2 = list()
```

Оба этих определения списка аналогичны - они создают пустой список.

Список необязательно должен содержать только однотипные объекты. Мы можем поместить в один и тот же список одновременно строки, числа, объекты других типов данных:

```
objects = [1, 2.6, "Hello", True]
```

Для проверки элементов списка можно использовать стандартную функцию **print**, которая выводит содержимое списка в удобочитаемом виде:

```
numbers = [1, 2, 3, 4, 5]
people = ["Tom", "Sam", "Bob"]
print(numbers)  # [1, 2, 3, 4, 5]
print(people)   # ["Tom", "Sam", "Bob"]
```

Конструктор **list** может принимать набор значений, на основе которых создается список:

```
numbers1 = [1, 2, 3, 4, 5]
numbers2 = list(numbers1)
print(numbers2)  # [1, 2, 3, 4, 5]
letters = list("Hello")
print(letters)      # ['H', 'e', 'l', 'l', 'o']
```

Обращение к элементам списка

Для обращения к элементам списка надо использовать индексы, которые представляют номер элемента в списка. Индексы начинаются с нуля. То есть первый элемент будет иметь индекс 0, второй элемент - индекс 1 и так далее. Для обращения к элементам с конца можно использовать отрицательные индексы, начиная с -1. То есть у последнего элемента будет индекс -1, у предпоследнего - -2 и так далее.

```
people = ["Tom", "Sam", "Bob"]
# получение элементов с начала списка
print(people[0])   # Tom
print(people[1])   # Sam
print(people[2])   # Bob

# получение элементов с конца списка
print(people[-2])   # Sam
print(people[-1])   # Bob
print(people[-3])   # Tom
```

Для изменения элемента списка достаточно присвоить ему новое значение:

```
people = ["Tom", "Sam", "Bob"]
people[1] = "Mike"  # изменение второго элемента
print(people[1])    # Mike
print(people)       # ["Tom", "Mike", "Bob"]
```

Перебор элементов

Для перебора элементов можно использовать как цикл **for**, так и цикл **while**.

Перебор с помощью цикла **for**:

```
people = ["Tom", "Sam", "Bob"]
for person in people:
    print(person)
```

Здесь будет производиться перебор списка **people**, и каждый его элемент будет помещаться в переменную **person**.

Перебор также можно сделать с помощью цикла **while**:

```
people = ["Tom", "Sam", "Bob"]
i = 0
while i < len(people):
    print(people[i])    # применяем индекс для получения элемента
    i += 1
```

Для перебора с помощью функции **len()** получаем длину списка. С помощью счетчика **i** выводит по элементу, пока значение счетчика не станет равно длине списка.

Получение части списка

Если необходимо получить какую-то определенную часть списка, то мы можем применять специальный синтаксис, который может принимать следующие формы:

list[:end]: через параметр **end** передается индекс элемента, до которого нужно копировать список

list[start:end]: параметр **start** указывает на индекс элемента, начиная с которого надо скопировать элементы

list[start:end:step]: параметр **step** указывает на шаг, через который будут копироваться элементы из списка. По умолчанию этот параметр равен 1.

```
people = ["Tom", "Bob", "Alice", "Sam", "Tim", "Bill"]

slice_people1 = people[:3]   # с 0 по 3
print(slice_people1)   # ["Tom", "Bob", "Alice"]

slice_people2 = people[1:3]   # с 1 по 3
print(slice_people2)   # ["Bob", "Alice"]

slice_people3 = people[1:6:2]   # с 1 по 6 с шагом 2
print(slice_people3)   # ["Bob", "Sam", "Bill"]
```

Можно использовать отрицательные индексы, тогда отсчет будет идти с конца, например, -1 - предпоследний, -2 - третий сконца и так далее.

```
people = ["Tom", "Bob", "Alice", "Sam", "Tim", "Bill"]

slice_people1 = people[:-1]   # с предпоследнего по нулевой
print(slice_people1)   # ["Tom", "Bob", "Alice", "Sam", "Tim",
"Bill"]

slice_people2 = people[-3:-1]   # с третьего с конца по
предпоследний
print(slice_people2)   # [ "Sam", "Tim"]
```

Кортежи

Кортеж (**tuple**) представляет последовательность элементов, которая во многом похожа на список за тем исключением, что кортеж является неизменяемым (**immutable**) типом. Поэтому мы не можем добавлять или удалять элементы в кортеже, изменять его.

Для создания кортежа используются круглые скобки, в которые помещаются его значения, разделенные запятыми:

```
tom = ("Tom", 23)
print(tom)     # ("Tom", 23)
```

Также для определения кортежа мы можем просто перечислить значения через запятую без применения скобок:

```
tom = "Tom", 23
print(tom)     # ("Tom", 23)
```

Если вдруг кортеж состоит из одного элемента, то после единственного элемента кортежа необходимо поставить запятую:

```
tom = ("Tom",)
```

Для создания кортежа из другого набора элементов, например, из списка, можно передать список в функцию **tuple(),** которая возвратит кортеж:

```
data = ["Tom", 37, "Google"]
tom = tuple(data)
print(tom)      # ("Tom", 37, "Google")
```

С помощью встроенной функции **len()** можно получить длину кортежа:

```
tom = ("Tom", 37, "Google")
print(len(tom))     # 3
```

Обращение к элементам кортежа

Обращение к элементам в кортеже происходит также, как и в списке, по индексу. Индексация начинается также с нуля при получении элементов с начала списка и с -1 при получении элементов с конца списка:

```
tom = ("Tom", 37, "Google", "software developer")
print(tom[0])       # Tom
print(tom[1])       # 37
print(tom[-1])      # software developer
```

Но так как кортеж - неизменяемый тип (**immutable**), то мы не сможем изменить его элементы. То есть следующая запись работать не будет:

```
tom[1] = "Tim"
```

Требования к выполнению лабораторной работы:

- **Запрещено** использовать словари и множества
- **Разрешается** использовать только функции **split, join, len, clear**, copy и функции явного приведения типов
- Собственные функции использовать **запрещено**

Варианты заданий:

1. Ввести строку, вывести на экран только слова, имеющие заданную длину.

2. Ввести строку и образец поиска. Найти позиции в строке, совпадающие с образцом и вывести их.

3. Ввести строку, вывести на экран только слова с симметричным расположением букв.

4. Ввести строку, подсчитать количество слов заданной длины.

5. Ввести строку, вывести пословно на экран, но слова в обратном порядке.

6. Ввести строку, заменить интервалы между словами на 2 пробела.

7. Ввести строку, выяснить, нет ли повторяющихся слов, и вывести их, если они есть. Можно считать, что все слова имеют длину ровно 3 символа.

8. Ввести строку, вывести на экран только слова начинающиеся с гласной буквы.

9. Ввести строку, вывести пословно на экран.

10. Ввести строку и слово, вывести индексы слов, которые при перестановке букв наоборот совпадают с введённым словом

11. Ввести строку, вывести самое длинное слово.

12. Ввести строку, вывести слово, содержащее наибольшее количество гласных букв.

13. Ввести строку, вывести самое короткое слово.

14. Ввести строку и два слова, заменить все вхождения первого слова на второе.

15. Ввести строку, подсчитать частоту появления каждой гласной буквы.

16. Ввести строку и слово, удалить все вхождения слова и вывести строку.

17. Ввести строку, вывести на экран только слова начинающиеся с согласной буквы.

18. Ввести строку, вывести слова по алфавиту (по первой букве).

19. Ввести строку и букву, вывести только слова, начинающиеся с заданной буквы.

20. Ввести строку и букву, вывести только слова, заканчивающиеся на заданную букву.

21. Ввести строку, вывести только слова, заканчивающиеся на гласную букву.

22. Ввести строку, вывести только слова, оканчивающиеся на согласную букву.

23. Ввести строку, вывести среднее по длине слово.

24. Ввести строку, вывести индексы повторяющихся слов

25. Ввести строку, вывести слова упорядоченные по убыванию по количеству имеющихся символов

Лабораторная работа 4: Функции и модули

Цель работы: Изучить функции, модули, сортировки. Реализовать приложение, импортируя необходимые функции из созданных модулей.

Вопросы, изучаемые в работе:

- Определение и использование функций
- Параметры функций и возвращаемые значения
- Импорт и использование модулей Python

Функция – это мини-программа внутри основной программы. Код такой подпрограммы отвечает за решение определенной задачи: например, в игре Тетрис будут отдельные функции для подсчета очков, рисования игрового поля, движения фигурки и так далее. Использование функций позволяет:

– Ограничить область видимости переменных функциями, которые их используют;

– Исключить дублирование кода;

– Разбить большую и сложную программу на небольшие мини-программы, которые можно вызывать в нужный момент;

– Выстроить простую и понятную структуру программы – такой код удобнее дебажить и поддерживать.

– У функций есть несколько особенностей:

– Функция выполняется только тогда, когда ее вызывает основная программа.

– В функцию можно передавать различные данные. Параметры – это переменные, которые используются при объявлении функции, аргументы – фактические значения, которые передаются переменным при вызове функции.

– Функции могут передавать результаты своей работы в основную программу или в другие функции.

Функции в Python

Python работает со встроенными и пользовательскими функциями. Встроенные функции – это уже знакомые нам `print(), input(), map(), zip()` и так далее. Пользовательские функции, в свою очередь, делятся на:

Рекурсивные (вызывают сами себя до тех пор, пока не будет достигнут нужный результат).

Анонимные, или лямбда-функции (объявляются в любом участке кода и сразу же вызываются).

Все остальные функции, которые определены пользователем и не относятся ни к рекурсивным, ни к анонимным.

В этой статье мы рассмотрим пользовательские функции с различными типами параметров, а в последующих статьях разберем анонимные и рекурсивные функции.

Объявление и вызов функций в Python

Для создания функции используют ключевое слово **`def`**. Вот пример простейшей функции, которая не получает и не возвращает никаких данных – просто выполняет одну команду по выводу строки с приветствием:

```
def my_function():
     print('Привет от Python')
```

Для вызова такой функции достаточно написать ее название:

```
my_function()
```

Результат вызова:

```
Привет от Python
```

А это пример простейшей функции с параметром:

```
def my_function(name):
     print(f'Привет, {name}')
```

При вызове функция получает аргумент:

```
my_function('Вася')
```

Результат вызова:

```
Привет, Вася
```

При вызове функция ожидает получить набор значений, соответствующий числу параметров. К примеру, эта функция должна получить при вызове два позиционных аргумента – имя и фамилию:

```
def my_function(name, lastname):
     print(f'Добрый день, {name} {lastname}')
```

Если передать в функцию два аргумента – **`my_function('Егор', 'Куликов')`**, результат вызова будет таким:

```
Добрый день, Егор Куликов
```

Но если число аргументов окажется меньше числа параметров – **`my_function('Алена')`**, возникнет ошибка:

```
my_function('Алена')
TypeError: my_function() missing 1 required positional argument:
'lastname'
```

Порядок обработки позиционных аргументов

Python обрабатывает позиционные аргументы слева направо:

```
def my_function(name, last_name, occupation, age):
     print(f'Сотрудник #1 - {name} {last_name} {occupation}
{age}')


info1, info2, info3, info4 = 'Алиса', 'Селезнева', 'скрам-мастер',
30
```

```
my_function(info1, info2, info3, info4)
my_function(info2, info3, info1, info4)
my_function(info4, info1, info2, info3)
```

Вывод:

```
Сотрудник #1 - Алиса Селезнева скрам-мастер 30
Сотрудник #1 - Селезнева скрам-мастер Алиса 30
Сотрудник #1 - 30 Алиса Селезнева скрам-мастер
```

Аргументы по умолчанию

Функция может использовать аргументы по умолчанию – они указываются после позиционных:

```
def my_function(strt, build, ap, city='Москва'):
     print(f'Адрес: г.{city}, ул.{strt}, д.{build}, кв.{ap}')
my_function('Красная', '5', '3', 'Тула')
my_function('Красная', '5', '3')
Результат:
Адрес: г.Тула, ул.Красная, д.5, кв.3
Адрес: г.Москва, ул.Красная, д.5, кв.3
```

Именованные аргументы

Помимо позиционных, в функцию можно передать именованные аргументы, причем порядок передачи именованных аргументов при вызове функции может не совпадать с порядком параметров:

```
def sales_price(price, discount=5):
    return price - price * discount / 100


print(sales_price(5000))
print(sales_price(5000, discount=10))
```

```
print(sales_price(discount=15, price=5000))
```

Вывод:

```
4750.0
4500.0
4250.0
```

Произвольное количество позиционных аргументов ***args**

До сих пор мы передавали в функцию определенное, заранее известное число позиционных аргументов. Если в функцию нужно передать произвольное количество аргументов, используют ***args:**

```
def my_function(*args):
    print(f'Минимальное число: {min(args)}, максимальное: {max(args)}')
my_function(1, 4, 5, 2, -5, 0, 12, 11)
```

Результат вызова:

```
Минимальное число: -5, максимальное: 12
```

При использовании ***args** функция получает кортеж аргументов, и к ним можно обращаться так же, как к элементам кортежа:

```
def my_function(*args):
    print(f'Первое слово: {args[0]}, последнее слово: {args[-1]}')
my_function('яблоко', 'виноград', 'апельсин', 'арбуз', 'слива', 'груша')
```

Результат вызова:

Аргументы ***args** обрабатываются после позиционных, но до аргументов по умолчанию:

```
def my_function(x, y, *args, kx=15, ky=15):
    print(x, y, args, kx, ky)
my_function(5, 6, 7, 8, 9, 0, 4)
```

Вывод:

```
5 6 (7, 8, 9, 0, 4) 15 15
```

Произвольное количество именованных аргументов ****kwargs**

Как уже было отмечено выше, именованные аргументы передаются в функцию в виде пар ключ=значение:

```
def my_function(cat1, cat2, cat3):
    print(f'Младший кот: {cat1}, старший кот: {cat2}')
my_function(cat1='Том', cat2='Барсик', cat3='Полосатик')
```

Результат вызова:

```
Младший кот: Том, старший кот: Барсик
```

В приведенном выше примере количество именованных аргументов известно заранее. Если в функцию нужно передать произвольное количество пар ключ=значение, используют параметр ****kwargs**. С ****kwargs** работают все методы словарей:

```
def my_function(**kwargs):
    print(f'Самый легкий металл - {min(kwargs, key=kwargs.get)} {min(kwargs.values())}, самый тяжелый - {max(kwargs, key=kwargs.get)} {max(kwargs.values())}')
my_function(осмий=22.61, цинк=7.1, золото=19.3, ртуть=13.6, олово=7.3)
```

Результат вызова:

```
Самый легкий металл - цинк 7.1, самый тяжелый - осмий 22.61
```

Аргументы типа ****kwargs** обрабатываются после позиционных, ***args** и аргументов по умолчанию:

```
def my_function(x, y, *args, kx=15, ky=15, **kwargs):
    print(x, y, args, kx, ky, kwargs)
my_function(7, 8, 0, 3, 4, 1, 8, 9, север=15, запад=25, восток=45, юг=10)
```

Вывод:

```
         7 8 (0, 3, 4, 1, 8, 9) 15 15 {'север': 15, 'запад': 25, 'восток': 45, 'юг': 10}
```

Передача аргументов в виде списка

Помимо кортежей и словарей, в функции можно передавать списки:

```
def my_function(stationery):
    for i, j in enumerate(stationery):
        print(f'Товар #{i + 1} - {j}')


stuff = ['карандаш', 'ручка', 'блокнот', 'альбом', 'тетрадь', 'ластик']
my_function(stuff)
```

Результат вызова:

```
        Товар #1 - карандаш
Товар #2 - ручка
Товар #3 - блокнот
Товар #4 - альбом
```

```
Товар #5 - тетрадь
Товар #6 - ластик
```

Функции с возвратом значений

Как уже было показано выше, функции могут получать и обрабатывать любые типы данных – строки, числа, списки, кортежи, словари. Результат обработки можно получить с помощью оператора **return**. Эта функция возвращает произведение произвольного количества значений:

```
def my_function(*args):
    prod = 1
    for i in args:
        prod *= i
    return prod
print(my_function(5, 6, 3, 11))
```

Значения передаются в функцию при вызове – **print(my_function(5, 6, 3, 11))**. Результат при таком наборе цифр будет равен 990. Оператор return может возвращать любое количество значений, причем значения возвращаются в виде кортежа:

```
def calculations(a, b):
    summa = a + b
    diff = a - b
    mul = a * b
    div = a / b
    return summa, diff, mul, div
num1, num2 = int(input()), int(input())
summa, diff, mul, div = calculations(num1, num2)
print(
    f'Сумма: {summa}\n'
    f'Разница: {diff}\n'
```

```
    f'Произведение: {mul}\n'
    f'Результат деления: {div:.2f}\n'
    )
```

Пример ввода:

```
49
6
```

Вывод:

```
Сумма: 55
Разница: 43
Произведение: 294
Результат деления: 8.17
```

Встроенные методы сортировки в Python

Стандартный метод сортировки списка по возрастанию – **`sort()`**. Пример использования:

```
nums = [54, 43, 3, 11, 0]
nums.sort()
print(nums) # Выведет [0, 3, 11, 43, 54]
```

Метод **`sorted()`** создает новый отсортированный список, не изменяя исходный. Пример использования:

```
nums = [54, 43, 3, 11, 0]
nums2 = sorted(nums)
print(nums, nums2) # Выведет [54, 43, 3, 11, 0] [0, 3, 11, 43, 54]
```

Если нам нужна сортировка от большего числа к меньшему, то установим флаг **`reverse=True`**. Примеры:

```
nums = [54, 43, 3, 11, 0]
nums.sort(reverse=True)
print(nums) # Выведет [54, 43, 11, 3, 0]
```

```
nums = [54, 43, 3, 11, 0]
nums2 = sorted(nums, reverse=True)
print(nums, nums2) # Выведет [54, 43, 3, 11, 0] [54, 43, 11, 3, 0]
```

Пузырьковая сортировка

Алгоритм попарно сравнивает элементы списка, меняя их местами, если это требуется. Он не так эффективен, если нам нужно сделать только один обмен в списке, так как данный алгоритм при достижении конца списка будет повторять процесс заново. Чтобы алгоритм не выполнялся бесконечно, мы вводим переменную, которая поменяет свое значение с **`True`** на **`False`**, если после запуска алгоритма список не изменился.

Сравниваются первые два элемента. Если первый элемент больше, то они меняются местами. Далее происходит все то же самое, но со следующими элементами до последней пары элементов в списке.

Пример пузырьковой сортировки:

```
def bubble(list_nums):
    swap_bool = True
    while swap_bool:
        swap_bool = False
        for i in range(len(list_nums) - 1):
            if list_nums[i] > list_nums[i + 1]:
                list_nums[i], list_nums[i + 1] = list_nums[i + 1],
list_nums[i]
                swap_bool = True
nums = [54, 43, 3, 11, 0]
bubble(nums)
```

```
print(nums) # Выведет [0, 3, 11, 43, 54]
```

Сортировка вставками

Алгоритм делит список на две части, вставляя элементы на их правильные места во вторую часть списка, убирая их из первой.

Если второй элемент больше первого, то оставляем его на своем месте. Если он меньше, то вставляем его на второе место, оставив первый элемент на первом месте. Далее перемещаем большие элементы во второй части списка вверх, пока не встретим элемент меньше первого или не дойдем до конца списка.

Пример сортировки вставками:

```
def insertion(list_nums):
    for i in range(1, len(list_nums)):
        item = list_nums[i]
        i2 = i - 1
        while i2 >= 0 and list_nums[i2] > item:
            list_nums[i2 + 1] = list_nums[i2]
            i2 -= 1
        list_nums[i2 + 1] = item
nums = [54, 43, 3, 11, 0]
insertion(nums)
print(nums) # Выведет [0, 3, 11, 43, 54]
```

Сортировка выборкой

Как и сортировка вставками, этот алгоритм в Python делит список на две части: основную и отсортированную. Наименьший элемент удаляется из основной части и переходит в отсортированную.

Саму отсортированную часть можно и не создавать, обычно используют крайнюю часть списка. И когда находится наименьший элемент списка, то переносим его на первое место, вставляя первый элемент на прошлое порядковое место наименьшего. Далее делаем все то же самое, но со следующим элементом, пока не достигнем конца списка.

Пример сортировки выборкой:

```
def selection(sort_nums):
    for i in range(len(sort_nums)):
        index = i
        for j in range(i + 1, len(sort_nums)):
            if sort_nums[j] < sort_nums[index]:
                index = j
        sort_nums[i], sort_nums[index] = sort_nums[index], 
sort_nums[i]
nums = [54, 43, 3, 11, 0]
selection(nums)
print(nums) # Выведет [0, 3, 11, 43, 54]
```

Модули

Модуль в языке Python представляет отдельный файл с кодом, который можно повторно использовать в других программах.

Для создания модуля необходимо создать собственно файл с расширением *.py, который будет представлять модуль. Название файла будет

представлять название модуля. Затем в этом файле надо определить одну или несколько функций.

Допустим, основной файл программы называется **main.py**. И мы хотим подключить к нему внешние модули.

Для этого сначала определим новый модуль: создадим в той же папке, где находится main.py, новый файл, который назовем **message.py**. Если используется PyCharm или другая IDE, то оба файла просто помещаются в один проект.

Соответственно модуль будет называться message. Определим в нем следующий код:

```
hello = "Hello all"
def print_message(text):
     print(f"Message: {text}")
```

Здесь определена переменная **hello** и функция **print_message**, которая в качестве параметра получает некоторый текст и выводит его на консоль.

В основном файле программы - **main.py** используем данный модуль:

```
import message      # подключаем модуль message

# выводим значение переменной hello
print(message.hello)        # Hello all
# обращаемся к функии print_message
message.print_message("Hello work")  # Message: Hello work
```

Для использования модуля его надо импортировать с помощью оператора **import**, после которого указывается имя модуля**: import message**.

Чтобы обращаться к функциональности модуля, нам нужно получить его пространство имен. По умолчанию оно будет совпадать с именем модуля, то есть в нашем случае также будет называться **message**.

Получив пространство имен модуля, мы сможем обратиться к его функциям по схеме пространство_имен.функция

Например, обращение к функции **print_message()** из модуля message:

```
message.print_message("Hello work")
```

И после этого мы можем запустить главный скрипт **main.py**, и он задействует модуль **message.py**. В частности, консольный вывод будет следующим:

```
Hello all
Message: Hello work
```

Подключение функциональности модуля в глобальное пространство имен

Другой вариант настройки предполагает импорт функциональности модуля в глобальное пространство имен текущего модуля с помощью ключевого слова **from**:

```
from message import print_message


# обращаемся к функии print_message из модуля message
print_message("Hello work")  # Message: Hello work


# переменная hello из модуля message не доступна, так как она не импортёирована
# print(message.hello)
# print(hello)
```

В данном случае мы импортируем из модуля message в глобальное пространство имен функцию `print_message()`. Поэтому мы сможем ее использовать без указания пространства имен модуля как если бы она была определена в этом же файле.

Требования к выполнению лабораторной работы:

- Тестирующая программа должна быть реализована в модуле `main.py`
- Списки и множества использовать **запрещено**

- Сторонние библиотеки использовать **запрещено**
- Необходимо реализовать собственную функцию сортировки в отдельном модуле
- **Самостоятельно изучить понятие O(n)** на примере сортировок из методического пособия. Иметь возможность объяснить на базовом уровне алгоритмическую сложность вашего алгоритма.

Варианты заданий:

1. Реализовать модуль, содержащий функцию, которая из двух массивов типа int, упорядоченных по убыванию, формирует новый массив двойной длины, уорядоченный по убыванию (слияние).
2. Реализовать модуль, содержащий функцию нахождения в массиве целых чисел элемента, ближайшего к значению второго аргумента типа int.
3. Реализовать модуль, содержащий функцию нахождения в массиве целых чисел наименьшего по абсолютной величине числа.
4. Реализовать модуль, содержащий функцию нахождения в массиве вещественных чисел числа с наименьшей дробной частью (дробная часть всегда положительна).
5. Реализовать модуль, содержащий функцию нахождения в массиве целых чисел разности индексов максимального и минимального элементов.
6. Реализовать модуль, содержащий функцию, которая в массиве вещественных чисел обнуляет все элементы, которые меньше среднего арифметического значения элементов исходного массива.
7. Реализовать модуль, содержащий функцию, которая вставляет в массив элемент с заданным индексом и заданным значением. Лишний элемент должен пропасть.
8. Реализовать модуль, содержащий функцию, которая удаляет из массива элемент с заданным индексом. Недостающий элемент должен быть обнулен.

9. Реализовать модуль, содержащий функцию, которая переставляет элементы массива типа int так, что все положительные элементы предшествуют отрицательным.

10. Реализовать модуль, содержащий функцию, которая переставляет элементы массива типа int так, что все четные значения предшествуют нечетным.

11. Реализовать модуль, содержащий функцию, находящую максимум из значений четырех аргументов типа float.

12. Реализовать модуль, содержащий функцию, которая удаляет из массива все элементы, являющиеся локальными минимумами. Локальным минимумом считается элемент, который меньше и своего левого соседа, и своего правого соседа. Недостающие элементы должны быть обнулены.

13. Реализовать модуль, содержащий функцию, которая возводит первый аргумент в степень, равную второму аргументу. Все значения имеют тип int.

14. Реализовать модуль, содержащий функцию, удаляющую лидирующие и заключительные пробелы и символы табуляции.

15. Реализовать модуль, содержащий функцию, которая переставляет элементы массива типа int так, что первое значения меняется с последним, второе - с предпоследним, и т.д. Общее количество обменов определяется вторым аргументом.

16. Реализовать модуль, содержащий функцию, проверяющую, является ли первый аргумент некоторой натуральной степенью второго аргумента. Все значения имеют тип int.

17. Реализовать модуль, содержащий функцию, находящую наименьшее общее кратное (НОК) двух чисел.

18. Реализовать модуль, содержащий функцию, которая удаляет из массива все элементы, являющиеся локальными максимумами. Локальным максимумом считается элемент, который больше и своего левого соседа, и своего правого соседа. Недостающие элементы должны быть обнулены.

19. Реализовать модуль, содержащий функцию, которая в массиве вещественных чисел обнуляет все элементы, которые больше среднего арифметического значения элементов исходного массива.

20. Реализовать модуль, содержащий функцию нахождения в массиве целых чисел наибольшего по абсолютной величине числа.

21. Реализовать модуль, содержащий функцию, которая переставляет элементы массива типа int так, что все отрицательные элементы предшествуют положительным.

22. Реализовать модуль, содержащий функцию, которая удаляет из массива элемент с заданным значением. Недостающий элемент должен быть обнулен.

23. Реализовать модуль, содержащий функцию, которая возвращает мажоритарный элемент. Мажоритарный элемент - это элемент, который встречается более ⌊n / 2⌋ раз.

24. Реализовать модуль, содержащий функцию нахождения в массиве целых чисел наибольшего по абсолютной величине числа.

25. Реализовать модуль, содержащий функцию, которая из двух массивов типа int, упорядоченных по возрастанию, формирует новый массив исключающий повторения чисел, упорядоченный по возрастанию

Лабораторная работа 5: Работа с вложенными списками.

Цель работы: Изучить вложенные списки. Реализовать с помощью вложенных списков приложение.

Вопросы, изучаемые в работе:

- Вложенные списки

– Способы работы с вложенными списками

Матрицами называются массивы элементов, представленные в виде прямоугольных таблиц, для которых определены правила математических действий. Элементами матрицы могут являться числа, алгебраические символы или математические функции.

В Python подобные таблицы можно представить в виде списка, элементы которого являются другими списками. Для примера создадим таблицу с тремя столбцами и тремя строками, заполненными произвольными буквами:

```
mas = [['й', 'ц', 'у'], ['к','е','н'], ['г', 'ш', 'щ']]
#Вывод всего двумерного массива
print(mas)
#Вывод первого элемента в первой строке
print(mas[0][0]) # Выведет й
#Вывод третьего элемента в третьей строке
print(mas[2][2]) # Выведет щ
```

Создание двумерных массивов

Создать такой массив в Python можно разными способами. Разберем первый:

```
# Создание таблицы с размером 3x3, заполненной нулями
a = 3
mas = [0] * a
for i in range(a):
    mas[i] = [0] * a
print(mas) # Выведет [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Второй способ предполагает создание пустого списка с добавлением в него новых списков. Рассмотрим на примере:

```
# Создание таблицы с размером 2x2, заполненной единицами
a = 2
mas = []
for i in range(a):
    mas.append([0] * a)
print(mas) # Выведет [[1, 1, 1], [1, 1, 1], [1, 1, 1]]
```

Третьим и самым простым способом является генератор списков с **x** строками, которые будут состоять из **y** элементов. Пример:

```
# Создание таблицы с размером 3x3, заполненной двойками
a = 3
mas = [[2] * a for i in range(a)]
print(mas) # Выведет [[2, 2, 2], [2, 2, 2], [2, 2, 2]]
```

Способы ввода двумерных массивов

Допустим, нам нужно ввести двумерный массив после запуска нашей программы. Для этого мы можем создать программу, которая будет построчно считывать значения нашего массива, а также количество строк в нем. Рассмотрим на примере:

```
a=int(input())
mas = []
for i in range(a):
    mas.append(list(map(int, input().split())))
print(mas)
```

Запускаем программу и сначала вводим количество строк в массиве (допустим, 3). Далее вводим строки в порядке их очереди. Например:

```
1 1 1
1 1 1
1 1 1
```

После этого данная программа выведет наш двумерный массив: **[[1, 1, 1], [1, 1, 1], [1, 1, 1]]**.

То же самое можно сделать с помощью генератора двумерных массивов:

```
mas = [list(map(int, input().split())) for i in
range(int(input()))]
# Вводим
3
1 1 1
1 1 1
1 1 1
print(mas) # Выведет [[1, 1, 1], [1, 1, 1], [1, 1, 1]]
```

Вывод двумерных массивов

Для обработки и вывода списков используются два вложенных цикла. Первый цикл – по порядковому номеру строки, второй – по ее элементам. Например, вывести массив можно так:

```
mas = [[1, 1, 1], [1, 1, 1], [1, 1, 1]]
for i in range(0, len(mas)):
```

```
    for i2 in range(0, len(mas[i])):
        print(mas[i][i2], end=' ')
    print()
# Выведет
1 1 1
1 1 1
1 1 1
```

То же самое можно сделать не по индексам, а по значениям массива:

```
mas = [[1, 1, 1], [1, 1, 1], [1, 1, 1]]
for i in mas:
    for i2 in i:
        print(i2, end=' ')
    print()
# Выведет
1 1 1
1 1 1
1 1 1
```

Способ с использованием метода **join():**

```
mas = [[1, 1, 1], [1, 1, 1], [1, 1, 1]]
for i in mas:
    print(' '.join(list(map(str, i))))
# Выведет
1 1 1
1 1 1
1 1 1
```

Вывод одной из строк двумерного массива можно осуществить с помощью цикла и того же метода **`join()`**. Для примера выведем вторую строку в произвольном двумерном массиве:

```
mas = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
string = 2
for i in mas[string-1]:
    print(i, end=' ')
# Выведет 1 1 1
```

Для вывода определенного столбца в двумерном массиве можно использовать такую программу:

```
mas = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
column = 2
for i in mas:
    print(i[column-1], end=' ')
# Выведет 2 5 8
```

Обработка двумерных массивов

Составим произвольный двумерный массив с числами и размерностью **4x4**:

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

Теперь поставим числа в каждой строке по порядку:

```
mas = [[2, 4, 7, 3], [4, 5, 6, 9], [1, 0, 4, 2], [7, 8, 4, 7]]
mas2 = []
for i in mas:
```

```
    mas2.append(sorted(i))
print(mas2)
# Выведет [[2, 3, 4, 7], [4, 5, 6, 9], [0, 1, 2, 4], [4, 7, 7, 8]]
```

А теперь расставим все числа по порядку, вне зависимости от их нахождения в определенной строке:

```
mas = [[2, 4, 7, 3], [4, 5, 6, 9], [1, 0, 4, 2], [7, 8, 4, 7]]
mas2 = []
for i in mas:
    for i2 in i:
        mas2.append(i2)
mas=sorted(mas2)
for x in range(0, len(mas), 4):
    e_c = mas[x : 4 + x]
    if len(e_c) < 4:
        e_c = e_c + [None for y in range(n - len(e_c))]
    print(list(e_c))
# Выведет
[0, 1, 2, 2]
[3, 4, 4, 4]
[4, 5, 6, 7]
[7, 7, 8, 9]
```

Требования к выполнению лабораторной работы:

- **Запрещено** использовать списки и множества
- **Запрещено** использовать сторонние библиотеки

Варианты заданий:

Вариант 1.

1. Вычислить сумму и число положительных элементов матрицы A[N, N], находящихся над главной диагональю.

2. Дана матрица B[N, M]. Найти в каждой строке матрицы максимальный и минимальный элементы и поменять их с первым и последним элементами строки соответственно.

Вариант 2.

1. Дана целая квадратная матрица n-го порядка. Определить, является ли она магическим квадратом, т. е. такой матрицей, в которой суммы элементов во всех строках и столбцах одинаковы.

2. Дана прямоугольная матрица A[N, N]. Переставить первый и последний столбцы местами и вывести на экран.

Вариант 3.

1. Определить, является ли заданная целая квадратная матрица n-го порядка симметричной (относительно главной диагонали).

2. Дана вещественная матрица размером n x m. Переставляя ее строки и столбцы, добиться того, чтобы наибольший элемент (или один из них) оказался в верхнем левом углу.

Вариант 4.

1. Дана прямоугольная матрица. Найти строку с наибольшей и строку с наименьшей суммой элементов. Вывести на печать найденные строки и суммы их элементов.

2. Дана квадратная матрица A[N, N], Записать на место отрицательных элементов матрицы нули, а на место положительных — единицы. Вывести на печать нижнюю треугольную матрицу в общепринятом виде.

Вариант 5.

1. Упорядочить по возрастанию элементы каждой строки матрицы размером n x m.

2. Дана действительная матрица размером n x m, все элементы которой различны. В каждой строке выбирается элемент с наименьшим значением. Если число четное, то заменяется нулем, нечетное - единицей. Вывести на экран новую матрицу.

Вариант 6.

1. Дана целочисленная квадратная матрица. Найти в каждой строке наибольший элемент и в каждом столбце наименьший. Вывести на экран.

2. Дана действительная квадратная матрица порядка N (N — нечетное), все элементы которой различны. Найти наибольший элемент среди стоящих на главной и побочной диагоналях и поменять его местами с элементом, стоящим на пересечении этих диагоналей.

Вариант 7.

1. Квадратная матрица, симметричная относительно главной диагонали, задана верхним треугольником в виде одномерного массива. Восстановить исходную матрицу и напечатать по строкам.

2. Для заданной квадратной матрицы сформировать одномерный массив из ее диагональных элементов. Найти след матрицы, просуммировав элементы одномерного массива. Преобразовать исходную матрицу по правилу: четные строки разделить на полученное значение, нечетные оставить без изменения.

Вариант 8.

1. Задана матрица порядка n и число к. Разделить элементы k-й строки на диагональный элемент, расположенный в этой строке.

2. Задана квадратная матрица. Получить транспонированную матрицу (перевернутую относительно главной диагонали) и вывести на экран.

Вариант 9.

1. Для целочисленной квадратной матрицы найти число элементов, кратных k, и наибольший из этих элементов.

2. В данной действительной квадратной матрице порядка n найти наибольший по модулю элемент. Получить квадратную матрицу порядка n — 1 путем отбрасывания из исходной матрицы строки и столбца, на пересечении которых расположен элемент с найденным значением.

Вариант 10.

1. Найти максимальный среди всех элементов тех строк заданной матрицы, которые упорядочены (либо по возрастанию, либо по убыванию).

2. Расположить столбцы матрицы D[M, N] в порядке возрастания элементов k-й строки (1 <= k <= M).

Вариант 11.

1. В данной действительной квадратной матрице порядка n найти сумму элементов строки, в которой расположен элемент с наименьшим значением. Предполагается, что такой элемент единственный.

2. Среди столбцов заданной целочисленной матрицы, содержащих только такие элементы, которые по модулю не больше 10, найти столбец с минимальным произведением элементов и поменять местами с соседним.

Вариант 12.

1. Для заданной квадратной матрицы найти такие k, что k-я строка матрицы совпадает с k-м столбцом.

2. Дана действительная матрица размером n x m. Требуется преобразовать матрицу: поэлементно вычесть последнюю строку из всех строк, кроме последней.

Вариант 13.

1. Определить наименьший элемент каждой четной строки матрицы A[M, N].

2. Найти наибольший и наименьший элементы прямоугольной матрицы и поменять их местами.

Вариант 14.

1. Задана квадратная матрица. Переставить строку с максимальным элементом на главной диагонали со строкой с заданным номером m.

2. Составить программу, которая заполняет квадратную матрицу порядка n натуральными числами 1, 2, 3, ..., n2, записывая их в нее «по спирали». Например, для n = 5 получаем следующую матрицу: 1 2 3 4 5 16 17 18 19 6 15 24 25 20 7 14 23 22 21 8 14 12 11 10 9

Вариант 15.

1. Определить номера строк матрицы R[M, N], хотя бы один элемент которых равен c, и элементы этих строк умножить на d.

2. Среди тех строк целочисленной матрицы, которые содержат только нечетные элементы, найти строку с максимальной суммой модулей элементов.

Вариант 16.

1. Создать программу, генерирующую случайную квадратную матрицу A[N, N] с целочисленными элементами. Вычислить произведение матрицы на ее транспонирование и вывести результат на экран.

2. Реализовать функцию, проверяющую, является ли данная квадратная матрица ортогональной, т.е. равна ли ее транспонирование ее инверсии.

Вариант 18.

1. Задав две матрицы A[M, N] и B[M, N], выполните сложение и вычитание матриц, чтобы вычислить результирующие матрицы C[M, N] и D[M, N].

2. Реализуйте функцию, которая перемножает две матрицы A[M, N] и B[N, P] для получения результирующей матрицы E[M, P]. Вывести результат на экран.

Вариант 19.

1. Создать программу, генерирующую случайную матрицу A[M, N] с целочисленными элементами. Вычислить и вывести на экран сумму всех элементов матрицы.

2. Реализовать функцию, которая находит определитель квадратной матрицы A[N, N] по формуле разложения Лапласа.

Вариант 20.

1. Задав две матрицы A[M, N] и B[N, M], реализуйте функцию, выполняющую умножение матриц для получения результирующей матрицы C[M, M]. Вывести результат на экран.

2. Вычислить след полученной матрицы C и вывести его на экран.

Вариант 21.

1. Задана квадратная матрица A[N, N], найдите сумму элементов в каждой строке и каждом столбце. Вывести суммы строк и столбцов.

2. Составить программу для проверки того, является ли квадратная матрица магическим квадратом, т.е. что сумма каждой строки, столбца и диагонали одинакова.

Вариант 22.

1. Написать программу для поиска строки с наибольшей и наименьшей суммой элементов в прямоугольной матрице. Вывести найденные строки с суммами их элементов.

2. Создать функцию, которая принимает на вход квадратную матрицу A[N, N], заменяет отрицательные элементы на нули, а положительные - на единицы и печатает нижнюю треугольную матрицу в обычном виде.

Вариант 23.

1. Найти сумму квадратов элементов каждой строки и каждого столбца в матрице A[N, N]. Вывести результат на экран.

2. Разделить каждый элемент матрицы A[M, N] на соответствующий элемент матрицы B[M, N], сохраняя результат в матрице C[M, N].

Вариант 24.

1. Реализовать программу, которая находит сумму среднего элемента в каждой строке и среднего элемента в каждом столбце целочисленной квадратной матрицы.

2. Создать функцию, которая берет вещественную квадратную матрицу нечетного порядка N с различными элементами, находит наибольший элемент на главной и побочной диагоналях и меняет его местами с элементом на их пересечении.

Вариант 25.

1. Напишите программу, которая восстанавливает квадратную матрицу, симметричную относительно главной диагонали, задавая верхний треугольник в виде одномерного массива. Вывести матрицу построчно.

2. Разработать функцию, которая принимает на вход квадратную матрицу, формирует одномерный массив из ее диагональных элементов, находит след матрицы путем суммирования этих элементов и преобразует исходную матрицу, деля четные строки на значение следа, а нечетные оставляя без изменений.

Лабораторная работа 6: Работа с файлами

Цель работы: Научиться работать с файлами в Python. Реализовать приложение.

Вопросы, изучаемые в работе:

- Чтение из файлов и запись в файлы в Python
- Режимы работы с файлами и файловые объекты
- Обработка исключений при работе с файлами

Для работы с файловой системой в Python используют модули **`os, os.path и shututil`**, а для операций с файлами – встроенные функции **`open(), close(), read(), readline(), write()`** и т. д. Прежде, чем мы перейдем к примерам использования конкретных методов, отметим один важный момент – корректный формат пути к файлам и каталогам.

Python считает некорректным стандартный для Windows формат: если указать путь к файлу в привычном виде **`'C:\Users\User\Python\letters.py'`**, интерпретатор вернет ошибку. Лучше всего указывать путь с помощью r-строк или с экранированием слэшей:

```
r'C:\Users\User\Python\letters.py'
'C:\\Users\\User\\Python\\letters.py'
```

Иногда также путь указывают с обратными слэшами:

```
'C:/Users/User/Python/letters.py'
```

Получение информации о файлах и директориях

Метод **getcwd()** возвращает путь к текущей рабочей директории в виде строки:

```
import os
os.getcwd()
>>> 'C:\\Users\\User\\Python'
```

С помощью метода **os.listdir()** можно получить список всех поддиректорий и файлов текущего рабочего каталога, при этом содержимое вложенных папок не отображается:

```
os.listdir()
>>> ['Data', 'lambda_functions.py', 'letters.py', 'os_methods.py',
'passw_generator.py', 'points.py', 'population.py']
```

Метод **os.walk()** возвращает генератор, в котором содержится вся информация о рабочем каталоге, включая содержимое всех поддиректорий:

```
import os
my_cwd = os.getcwd()
result = os.walk(my_cwd)
for i, j, k in result:
    for file in k:
        print(file)

>>> lambda_functions.py
>>> letters.py
>>> os_methods.py
>>> passw_generator.py
>>> points.py
```

```
>>> population.py
>>> books_to_process.txt
>>> challenge_data.txt
>>> ledger.txt
```

Много полезных методов содержится в модуле **os.path**. Так можно извлечь имя файла из полного пути:

```
os.path.basename(r'C:\Users\User\Python\letters.py')
>>> 'letters.py'
```

А так можно получить путь к директории **/** файлу, в который не включается собственно поддиректория или имя файла:

```
os.path.dirname(r'C:\Users\User\Python\Data')
>>> 'C:\\Users\\User\\Python'
```

Метод **path.isdir()** возвращает **True**, если переданная в метод директория существует, и **False** – в противном случае:

```
os.path.isdir(r'C:\Users\User\Python\Data\Samples')
>>> False
```

Операции с каталогами и файлами в Python

Для создания новых директорий служит **os.mkdir()**; в метод нужно передать полный путь, включающий название нового каталога:

```
import os
my_cwd = os.getcwd()
new_dir = 'Solutions'
path = os.path.join(my_cwd, new_dir)
```

```
os.mkdir(path)
print(os.listdir())
```

Результат:

```
 ['Data', 'lambda_functions.py', 'letters.py', 'os_methods.py',
'passw_generator.py', 'points.py', 'population.py', 'Solutions']
```

Создание директорий в Python

Для создания новых каталогов используют два метода:

`os.mkdir()` – аналог CLI команды **`mkdir`**; создает новую папку по указанному пути, при условии, что все указанные промежуточные (вложенные) директории уже существуют.

`os.makedirs()` – аналог CLI команды **`mkdir -p dir1\dir2`**; помимо создания целевой папки, создает все промежуточные директории, если они не существуют.

Пример использования **`os.mkdir():`**

```
import os
new_dir = 'NewProjects'
parent_dir = r 'C:\Users\User\Python'
path = os.path.join(parent_dir, new_dir)
os.mkdir(path)
print(f'Директория {new_dir} создана: {os.listdir()}')
```

Результат:

```
Директория NewProjects создана: ['Data', 'lambda_functions.py',
'letters.py', 'NewProjects', 'os_methods.py', 'Other',
'passw_generator.py', 'points.py', 'population.py', 'Solutions']
```

Копирование файлов и директорий в Python

Для копирования файлов используют метод **shutil.copy2()**, который принимает два аргумента – источник файла и директорию, в которую нужно скопировать файл:

```
import os

import shutil

dest_path = r'C:\Users\User\Python\Data'

source_path = r'C:\Users\User\lambda_exp.txt'

print(f'Файлы в директории {os.path.basename(dest_path)} до
копирования файла \

{os.path.basename(source_path)}: {os.listdir(dest_path)}\n')

copy_file = shutil.copy2(source_path, dest_path)

print(f'Файлы в директории {os.path.basename(dest_path)} после
копирования файла \

{os.path.basename(source_path)}: {os.listdir(dest_path)}')
```

Вывод:

```
Файлы в директории Data до копирования файла lambda_exp.txt:
['books_to_process.txt', 'challenge_data.txt', 'ledger.txt']

Файлы в директории Data после копирования файла lambda_exp.txt:
['books_to_process.txt', 'challenge_data.txt', 'lambda_exp.txt',
'ledger.txt']
```

Все содержимое каталога сразу можно скопировать с помощью **shutil.copytree():**

```
import os

import shutil

dest_path = r'C:\Users\User\Python\Other\Files'

source_path = r'C:\Users\User\Python\Other\Scripts'
```

```
print(f'Содержимое директории {os.path.basename(dest_path)} до копирования каталога \
{os.path.basename(source_path)}: {os.listdir(dest_path)}\n')
copy_dir = shutil.copytree(source_path, dest_path, dirs_exist_ok=True)
print(f'Содержимое директории {os.path.basename(dest_path)} после копирования \
{os.path.basename(source_path)}: {os.listdir(dest_path)}\n')
```

Вывод:

```
        Содержимое директории Files до копирования каталога Scripts: []
Содержимое директории Files после копирования Scripts: ['progression.py', 'sitemap_generator.py']
```

Удаление файлов и директорий

Для удаления директории вместе со всеми файлами используют **shutil.rmtree()**:

```
import os
import shutil
dir_path = r'C:\Users\User\Python\Other'
remove_dir = 'Files'
path = os.path.join(dir_path, remove_dir)
print(f'Содержимое директории {os.path.basename(dir_path)} до удаления каталога \
{remove_dir}: {os.listdir(dir_path)}\n')
shutil.rmtree(path)
print(f'Содержимое директории {os.path.basename(dir_path)} после удаления \
{remove_dir}: {os.listdir(dir_path)}\n')
```

Вывод:

```
Содержимое директории Other до удаления каталога Files: ['Files',
'Projects']

Содержимое директории Other после удаления Files: ['Projects']
```

Для удаления файлов используют метод **os.remove():**

```
import os

import shutil

dir_path = r'C:\Users\User\Python\Other\Scripts'

remove_file = 'tetris_game.py'

path = os.path.join(dir_path, remove_file)

print(f'Содержимое директории {os.path.basename(dir_path)} до
удаления файла \

{remove_file}: {os.listdir(dir_path)}\n')

os.remove(path)

print(f'Содержимое директории {os.path.basename(dir_path)} после
удаления \

{remove_file}: {os.listdir(dir_path)}\n')
```

Вывод:

```
Содержимое директории Scripts до удаления файла tetris_game.py:
['tetris_game.py']

Содержимое директории Scripts после удаления tetris_game.py: []
```

Работа с файлами в Python

Открыть файл для проведения каких-либо манипуляций можно двумя способами:

С помощью функции **open()** – в этом случае после завершения работы нужно будет закрыть файл с помощью **close():**

```
f = open('task.txt', 'a', encoding='utf-8')
f.write('\n2) Написать модуль авторизации')
f.close()
```

С использованием менеджера контекста **with**, который автоматически и самостоятельно закроет файл, когда надобность в нем отпадет:

```
with open('task.txt', 'a', encoding='utf-8') as f:
    f.write('\n2) Написать модуль авторизации')
```

Режимы доступа и записи

Таблица 8. Режимы работы файлов.

'r'	Открывает файл для чтения. Возвращает ошибку, если указанный файл не существует.
'w'	Открывает файл для записи, причем перезаписывает содержимое, если оно есть. Создает файл, если он не существует.
'a'	Открывает файл для записи и добавляет новую информацию, не перезаписывая существующую. Создает файл, если он не существует.
'w+'	Открывает файл для чтения и записи, перезаписывает содержимое.
'r+'	Открывает файл для чтения и дополнения, не перезаписывает содержимое.

'x'	Создает новый пустой файл. Возвращает ошибку, если файл с таким именем уже существует.

Методы работы с файлами

Для чтения данных используют **read().** Метод **read()** по умолчанию возвращает все содержимое файла:

```
with open('books.txt', 'r', encoding='utf-8') as f:
    info = f.read()
print(info)
```

Вывод:

```
1. "Террор", Дэн Симмонс
2. "Она же Грейс", Маргарет Этвуд
3. "Облачный атлас", Дэвид Митчелл
4. "Искупление", Иэн Макьюэн
5. "Госпожа Бовари", Гюстав Флобер
```

При необходимости объем выводимой информации можно ограничить определенным количеством символов:

```
with open('movies.txt', 'r', encoding='utf-8') as f:
    info = f.read(15)
print(info)
```

Вывод:

```
"Из машины"
```

Метод **readline()** позволяет считывать информацию из текстовых файлов построчно:

```
with open('books.txt', 'r', encoding='utf-8') as f:
    info = f.readline()
print(info)
```

Вывод:

```
"Террор", Дэн Симмонс
```

Для получения всех строк файла используют метод **readlines()**, который возвращает содержимое в виде списка – вместе со всеми спецсимволами:

```
with open('books.txt', 'r', encoding='utf-8') as f:
    info = f.readlines()
print(info)
```

Вывод:

```
 ['1. "Террор", Дэн Симмонс\n', '2. "Она же Грейс", Маргарет
Этвуд\n', '3. "Облачный атлас", Дэвид Митчелл\n', '4.
"Искупление", Иэн Макьюэн\n', '5. "Госпожа Бовари", Гюстав
Флобер']
```

Чтобы избавиться от лишних пробелов, символа новой строки (и любых других спецсимволов), используют методы **rstrip(), lstrip()** или **strip():**

```
with open('books.txt', 'r', encoding='utf-8-sig') as f:
    info = [line.strip() for line in f.readlines()]
print(info)
```

Вывод:

```
 ['1. "Террор", Дэн Симмонс', '2. "Она же Грейс", Маргарет Этвуд',
'3. "Облачный атлас", Дэвид Митчелл', '4. "Искупление", Иэн
Макьюэн', '5. "Госпожа Бовари", Гюстав Флобер']
```

Для записи информации в файл используют метод **write():**

```
with open('books.txt', 'a', encoding='utf-8') as f:
    f.write('\n6. "Война и мир", Лев Толстой\n')
```

Или **writelines():**

```
with open('books.txt', 'a', encoding='utf-8') as f:
    f.writelines(['7. "Преступление и наказание", Федор
Достоевский\n',
                  '8. "Мизери", Стивен Кинг\n',
                   '9. "Джейн Эйр", Шарлотта Бронте\n'])
```

Требования к выполнению лабораторной работы:

- **Запрещено** использовать словари и множества
- **Запрещено** использовать внешние библиотеки

Варианты заданий:

1. Написать программу для подсчёта количества строк, слов и символов в исходном текстовом файле и выводом собранной статистики в новый текстовый файл.

2. Написать программу для копирования слов имеющих заданную длину пользователем из одного текстового файла в другой.

4. Написать программу для копирования слов начинающихся с заглавной буквы из одного текстового файла в другой.

5. Написать программу для замены определенного слова или фразы в текстовом файле. Записать строки, включающую фразу в текстовый другой файл.

6. Написать программу для сортировки строк в текстовом файле по алфавиту и их записью в новый текстовый файл.

7. Написать программу для объединения нескольких текстовых файлов в один файл с возможностью сокращения размера выходного файла.

8. Написать программу для извлечения повторяющихся слов из одного файла и записи их в другой текстовый файл.

9. Написать программу разбиения большого текстового файла на меньшие файлы заданного размера.

10. Написать программу для вычисления среднего, минимального и максимального числовых значений из файла с любыми данными и их записью в новый текстовый файл.

11. Реализовать программу преобразования слов текстового файла из верхнего регистра в нижний или наоборот. Изменённые слова записать в новый текстовый файл

12. Написать программу для копирования слов из текстового файла, которые содержат заданные символы пользователем в новый текстовый файл.

13. Написать программу для копирования заданных пользователем слов из исходного текстового файла в новый текстовый файл.

14. Написать программу для создания списка файлов указанного каталога и его подкаталогов. Вывести в текстовом файле иерархию каталогов.

15. Написать программу для поиска самого большого файла в каталоге и самых больших файлов в его подкаталогах. Вывести в текстовом файле иерархию каталогов, найденные файлы выделить.

16. Написать программу для вычисления общего размера каталога и его подкаталогов. Вывести в текстовом файле иерархию каталогов с их размером.

17. Написать программу для поиска и удаления файлов старше заданного количества дней. Вывести в текстовом файле иерархию каталогов

18. Написать программу для сравнения двух текстовых файлов и записи различий в новый текстовый файл.

19. Написать программу для переименования файлов в каталоге по определенному шаблону. Вывести в текстовом файле иерархию каталогов, изменённые файлы выделить

20. Написать программу для удаления слов из текстового файла, имеющие заданные буквы пользователем. Вывести в новом текстовом файле удалённые слова.

21. Написать программу для кодирования исходного текстового файла и его раскодированной записи в новый текстовый файл.

Лабораторная работа 7: Структуры данных – Словари, множества. JSON.

Цель работы: Изучить словари, множества и JSON. Реализовать приложение, используя JSON для хранения актуальных данных.

Вопросы, изучаемые в работе:

- Знакомство со словарями и множествами в Python
- Методы и операции со словарями и множествами
- Использование словарей и множеств для хранения и поиска данных
- Использование JSON для хранения данных

Словари

Словарь (**dictionary**) в языке Python хранит коллекцию элементов, где каждый элемент имеет уникальный ключ и ассоциированое с ним некоторое значение.

Определение словаря имеет следующий синтаксис:

```
dictionary = { ключ1:значение1, ключ2:значение2, ....}
```

В фигурных скобках через запятую определяется последовательность элементов, где для каждого элемента сначала указывается ключ и через двоеточие его значение.

Определим словарь:

```
users = {1: "Tom", 2: "Bob", 3: "Bill"}
```

В словаре **users** в качестве ключей используются числа, а в качестве значений - строки. То есть элемент с ключом 1 имеет значение **"Tom"**, элемент с ключом 2 - значение **"Bob"** и т.д.

Другой пример:

```
emails = {"tom@gmail.com": "Tom", "bob@gmai.com": "Bob",
"sam@gmail.com": "Sam"}
```

В словаре emails в качестве ключей используются строки - электронные адреса пользователей и в качестве значений тоже строки - имена пользователей.

Но необязательно ключи и строки должны быть однотипными. Они могу представлять разные типы:

```
objects = {1: "Tom", "2": True, 3: 100.6}
```

Получение и изменение элементов

Для обращения к элементам словаря после его названия в квадратных скобках указывается ключ элемента:

```
dictionary[ключ]
```

Например, получим и изменим элементы в словаре:

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
```

```
# получаем элемент с ключом "+11111111"
print(users["+11111111"])      # Tom
```

```
# установка значения элемента с ключом "+33333333"
users["+33333333"] = "Bob Smith"
print(users["+33333333"])   # Bob Smith
```

Если при установки значения элемента с таким ключом в словаре не окажется, то произойдет его добавление:

```
users["+4444444"] = "Sam"
```

Но если мы попробуем получить значение с ключом, которого нет в словаре, то Python сгенерирует ошибку KeyError:

```
user = users["+4444444"]    # KeyError
```

И чтобы предупредить эту ситуацию перед обращением к элементу мы можем проверять наличие ключа в словаре с помощью выражения ключ **in** словарь. Если ключ имеется в словаре, то данное выражение возвращает **True**:

```
key = "+4444444"
if key in users:
    user = users[key]
    print(user)
else:
    print("Элемент не найден")
```

Также для получения элементов можно использовать метод **get**, который имеет две формы:

get(key): возвращает из словаря элемент с ключом **key**. Если элемента с таким ключом нет, то возвращает значение **None**

get(key, default): возвращает из словаря элемент с ключом **key**. Если элемента с таким ключом нет, то возвращает значение по умолчанию **default**

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}

user1 = users.get("+55555555")
print(user1)    # Alice
```

```
user2 = users.get("+33333333", "Unknown user")
print(user2)    # Bob
user3 = users.get("+44444444", "Unknown user")
print(user3)    # Unknown user
Удаление
Для удаления элемента по ключу применяется оператор del:
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}

del users["+55555555"]
print(users)    # { "+11111111": "Tom", "+33333333": "Bob"}
```

Но стоит учитывать, что если подобного ключа не окажется в словаре, то будет выброшено исключение KeyError. Поэтому опять же перед удалением желательно проверять наличие элемента с данным ключом.

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}

key = "+55555555"
if key in users:
    del users[key]
    print(f"Элемент с ключом {key} удален")
else:
    print("Элемент не найден")
```

Другой способ удаления представляет метод **pop()**. Он имеет две формы:

pop(key): удаляет элемент по ключу **key** и возвращает удаленный элемент. Если элемент с данным ключом отсутствует, то генерируется исключение KeyError

pop(key, default): удаляет элемент по ключу **key** и возвращает удаленный элемент. Если элемент с данным ключом отсутствует, то возвращается значение **default**

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
key = "+55555555"
user = users.pop(key)
print(user)     # Alice

user = users.pop("+4444444", "Unknown user")
print(user)     # Unknown user
```

Если необходимо удалить все элементы, то в этом случае можно воспользоваться методом **clear():**

```
users.clear()
```

Копирование и объединение словарей

Метод **copy()** копирует содержимое словаря, возвращая новый словарь:

```
users = {"+1111111": "Tom", "+3333333": "Bob", "+5555555":
"Alice"}
students = users.copy()
```

```
print(students)     # {"+1111111": "Tom", "+3333333": "Bob",
"+5555555
```

Метод **update()** объединяет два словаря:

```
users = {"+1111111": "Tom", "+3333333": "Bob"}


users2 = {"+2222222": "Sam", "+6666666": "Kate"}
users.update(users2)


print(users)    # {"+1111111": "Tom", "+3333333": "Bob",
"+2222222": "Sam", "+6666666": "Kate"}
print(users2)   # {"+2222222": "Sam", "+6666666": "Kate"}
```

При этом словарь **users2** остается без изменений. Изменяется только словарь users, в который добавляются элементы другого словаря. Но если необходимо, чтобы оба исходных словаря были без изменений, а результатом объединения был какой-то третий словарь, то можно предварительно скопировать один словарь в другой:

```
users3 = users.copy()
users3.update(users2)
```

Перебор словаря

Для перебора словаря можно воспользоваться циклом for:

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
for key in users:
    print(f"Phone: {key}  User: {users[key]} ")
```

При переборе элементов мы получаем ключ текущего элемента и по нему можем получить сам элемент.

Другой способ перебора элементов представляет использование метода **items():**

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
for key, value in users.items():
    print(f"Phone: {key}  User: {value} ")
```

Метод **items()** возвращает набор кортежей. Каждый кортеж содержит ключ и значение элемента, которые при переборе мы тут же можем получить в переменные **key** и **value**.

Также существуют отдельно возможности перебора ключей и перебора значений. Для перебора ключей мы можем вызвать у словаря метод **keys():**

```
for key in users.keys():
    print(key)
```

Правда, этот способ перебора не имеет смысла, так как и без вызова метода **keys()** мы можем перебрать ключи, как было показано выше.

Для перебора только значений мы можем вызвать у словаря метод **values()**:

```
for value in users.values():
    print(value)
```

Вложенные словари

Кроме простейших объектов типа чисел и строк словари также могут хранить и более сложные объекты - те же списки, кортежи или другие словари:

```
users = {
    "Tom": {
        "phone": "+971478745",
        "email": "tom12@gmail.com"
    },
    "Bob": {
        "phone": "+876390444",
        "email": "bob@gmail.com",
        "skype": "bob123"
    }
}
```

В данном случае значение каждого элемента словаря в свою очередь представляет отдельный словарь.

Для обращения к элементам вложенного словаря соответственно необходимо использовать два ключа:

```
old_email = users["Tom"]["email"]
users["Tom"]["email"] = "supertom@gmail.com"
print(users["Tom"])     # { phone": "+971478745", "email":
"supertom@gmail.
```

Но если мы попробуем получить значение по ключу, который отсутствует в словаре, Python сгенерирует исключение KeyError:

```
tom_skype = users["Tom"]["skype"]   # KeyError
```

Во всем остальном работа с комплексными и вложенными словарями аналогична работе с обычными словарями.

Множества

Множество (**set**) представляют еще один вид набора, который хранит только уникальные элементы. Для определения множества используются фигурные скобки, в которых перечисляются элементы:

```
users = {"Tom", "Bob", "Alice", "Tom"}
print(users)    # {"Alice", "Bob", "Tom"}
```

Обратите внимание, что несмотря на то, что функция print вывела один раз элемент **"Tom"**, хотя в определении множества этот элемент содержится два раза. Все потому что множество содержит только уникальные значения.

Также для определения множества может применяться функция **set()**, в которую передается список или кортеж элементов:

```
people = ["Mike", "Bill", "Ted"]
users = set(people)
print(users)    # {"Mike", "Bill", "Ted"}
```

Функцию set удобно применять для создания пустого множества:

```
users = set()
```

Для получения длины множества применяется встроенная функция **len()**:

```
users = {"Tom", "Bob", "Alice"}
```

```
print(len(users))       # 3
```

Добавление элементов

Для добавления одиночного элемента вызывается метод **add():**

```
users = set()
users.add("Sam")
print(users)
```

Удаление элементов

Для удаления одного элемента вызывается метод **remove()**, в который передается удаляемый элемент. Но следует учитывать, что если такого элемента не окажется в множестве, то будет сгенерирована ошибка. Поэтому перед удалением следует проверять на наличие элемента с помощью оператора **in**:

```
users = {"Tom", "Bob", "Alice"}

user = "Tom"
if user in users:
    users.remove(user)
print(users)    # {"Bob", "Alice"}
```

Также для удаления можно использовать метод **discard(),** который не будет генерировать исключения при отсутствии элемента:

```
users = {"Tom", "Bob", "Alice"}

users.discard("Tim")    # элемент "Tim" отсутствует, и метод
ничего не делает
print(users)    #  {"Tom", "Bob", "Alice"}
```

```
users.discard("Tom")    # элемент "Tom" есть, и метод удаляет
элемент
print(users)    #  {"Bob", "Alice"}
```

Для удаления всех элементов вызывается метод **clear()**:

```
users.clear()
```

Перебор множества

Для перебора элементов можно использовать цикл **for**:

```
users = {"Tom", "Bob", "Alice"}

for user in users:
    print(user)
```

При переборе каждый элемент помещается в переменную **user**.

JSON (JavaScript Object Notation) - простой формат обмена данными, основанный на подмножестве синтаксиса JavaScript. Модуль json позволяет кодировать и декодировать данные в удобном формате.

Кодирование основных объектов Python:

```
>>>
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
```

```
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
```

Компактное кодирование:

```
>>>
>>> import json
>>> json.dumps([1,2,3,{'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

Красивый вывод:

```
>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Декодирование (парсинг) JSON:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
```

Основы

json.dump - сериализует obj как форматированный JSON поток в fp.

Ключи в парах ключ/значение в JSON всегда являются строками. Когда словарь конвертируется в JSON, все ключи словаря преобразовываются в строки. В результате этого, если словарь сначала преобразовать в JSON, а потом обратно в словарь, то можно не получить словарь, идентичный исходному. Другими словами, **loads(dumps(x)) != x**, если **x** имеет нестроковые ключи.

json.load - десериализует JSON из **fp**.

Если не удастся десериализовать JSON, будет возбуждено исключение ValueError.

json.loads - десериализует **s** (экземпляр **str**, содержащий документ JSON) в объект Python.

Остальные аргументы аналогичны аргументам в **load().**

Кодировщики и декодировщики

Класс **json.JSONDecoder -** простой декодер JSON.

Выполняет следующие преобразования при декодировании (смотри таблицу 9):

Таблица 9. Соответствие типов при декодировании.

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True

JSON	Python
false	False
null	None

Он также понимает **`NaN`**, **`Infinity`**, и **`-Infinity`** как соответствующие значения **`float`**, которые находятся за пределами спецификации JSON.

Класс **`json.JSONEncoder`**

Расширяемый кодировщик JSON для структур данных Python. Поддерживает следующие объекты и типы данных по умолчанию (смотри таблицу 10):

Таблица 10. Соответствие типов при кодировании.

Python	JSON
dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null

Требования к выполнению лабораторной работы:

- **Запрещено** использовать внешние библиотеки

Варианты заданий:

1. Разработать систему, реализующую CRUD операции с базой данных школы.

2. Разработать систему, реализующую CRUD операции с базой данных университета.

3. Разработать систему, реализующую CRUD операции с базой данных фондового рынка.

4. Разработать систему, реализующую CRUD операции с базой данных театра

5. Разработать систему, реализующую CRUD операции с базой данных автошколы.

6. Разработать систему, реализующую CRUD операции с базой данных метеорологической компании.

7. Разработать систему, реализующую CRUD операции с базой данных магазина продуктовых товаров.

8. Разработать систему, реализующую CRUD операции с базой данных магазина компьютерных товаров.

9. Разработать систему, реализующую CRUD операции с базой данных аптеки.

10. Разработать систему, реализующую CRUD операции с базой данных кинотеатра.

11. Разработать систему, реализующую CRUD операции с базой данных магазина детских игрушек.

12. Разработать систему, реализующую CRUD операции с базой данных психологического центра.

13. Разработать систему, реализующую CRUD операции с базой данных больницы.

14. Разработать систему, реализующую CRUD операции с базой данных аэропорта.

15. Разработать систему, реализующую CRUD операции с базой данных торгового центра.

16. Разработать систему, реализующую CRUD операции с базой данных ЖД вокзала.

17. Разработать систему, реализующую CRUD операции с базой данных социальной сети.

18. Разработать систему, реализующую CRUD операции с базой данных магазина автозапчастей.

19. Разработать систему, реализующую CRUD операции с базой данных жилого комплекса.

20. Разработать систему, реализующую CRUD операции с базой данных книжного магазина.

21. Разработать систему, реализующую CRUD операции с базой данных музыкальной школы.

22. Разработать систему, реализующую CRUD операции с базой данных ресторана.

23. Разработать систему, реализующую CRUD операции с базой данных музея.

24. Разработать систему, реализующую CRUD операции с базой данных отеля.

25. Разработать систему, реализующую CRUD операции с базой данных цветочного магазина.

Лабораторная работа 8: Модули и библиотеки

Цель работы: Научиться работать с внешними библиотеками и фреймворками в Python. Реализовать приложение.

Вопросы, изучаемые в работе:

- Работа с внешними модулями и библиотеками в Python
- Установка и импорт пакетов
- Использование популярных библиотек для решения конкретных задач (например, NumPy, Pandas, Matplotlib).

Лабораторные работы выдаются лично преподавателем