1. Вступление

Представьте, что перед вами поставлена задача: даны анкеты учащихся некоторого класса и требуется подсчитать количество мальчиков и девочек в этом классе.

Можно предложить следующий способ решения. На школьной доске отведём место для двух чисел – количества мальчиков и количества девочек, и для начала в этих местах запишем по нулю. Далее будем перебирать анкету за анкетой и смотреть, какой пол указан в просматриваемой анкете. Если пол мужской, то будем заменять число на доске, показывающее количество мальчиков, числом, большим на единицу. Если же пол женский, то такую операцию будем выполнять с числом на доске, показывающим количество девочек. Как только мы переберём все анкеты, на доске останутся два числа, равные количеству мальчиков и девочек в классе.

В данном примере отражены следующие особенности программирования.

- Чтобы решить задачу, необходимо придумать метод, который за конечное количество действий приводит к решению (либо к осознанию, что решения нет). Такой метод называется *алгоритмом*.

- При решении задачи приходится работать с данными, которые имеют свой *тип*. К примеру, данными являются переменные *числа*, записанные на доске, а также *упорядоченное множество* анкет, содержащих *текстовую* информацию о том, является ли ученик мальчиком или девочкой.

- В ходе работы алгоритма иногда приходится повторить одно и то же действие несколько раз. Это называется *циклом*. В нашем примере в цикле просматриваются анкеты учеников и меняются числа на доске.

- В некоторых случаях выполняемые действия зависят от тех или иных условий. Такая алгоритмическая конструкция называется *ветвлением*. В примере, приведённом выше, изменение первого или второго числа на доске зависело от записи в просматриваемой анкете.

- Наконец заметим, что алгоритм, приведённый в примере, осуществлял *исполнитель* – человек. Поэтому очень важно, на каком языке записан алгоритм. Компьютеры «понимают» только язык машинных кодов. Но с машинными кодами человеку очень трудно иметь дело. Поэтому были созданы *языки программирования*, которые позволяют по особым правилам с помощью словесных конструкций записать алгоритм и сохранить так называемый *исходный код* программы в обычном текстовом файле. Далее этот файл обрабатывается или *транслируется* с помощью специальной программы, которая либо создаёт файл машинных кодов, который уже выполняется в операционной системе (в этом случае метод обработки исходного кода называется *компиляцией*), либо сама выполняет записанные в текстовом файле команды друг за дружкой и таким образом возвращает некоторый результат (в данном случае метод обработки исходного кода называется *интерпретацией*).

Надеемся, что после данного рассуждения становятся понятны слова Никлауса Вирта, создателя языка программирования pascal:

алгоритмы + структуры данных = программы.

Далее мы рассмотрим язык программирования python. Программы, записанные на этом языке, являются *интерпретируемыми*.

Для написания исходных текстов программ нужно скачать и установить бесплатные интегрированные среды программирования:

- либо официальный релиз Python и IDLE,
- либо Thonny,
- либо Python + Jupyter Notebook.

Наиболее простая IDE – Thonny. Её можно скачать по ссылке:
`https://github.com/thonny/thonny/releases/download/v3.2.7/thonny-3.2.7.exe`

2. Основы языка Python

2.1. Встроенные типы данных. Простейший ввод-вывод

2.1.1. Встроенные типы данных

В программном коде мы будем использовать следующие встроенные классы (типы) данных:

Тип данных	Пояснение
`int`	целое число (с произвольным количеством знаков)
`float`	число с плавающей точкой
`str`	строка
`bool`	логическая величина, принимающая значения `True` (истина) или `False` (ложь)
`tuple`	кортеж – упорядоченный (перенумерованный) набор, состоящий из разнотипных элементов
`range`	последовательность чисел, обычно используемая в циклах
`list`	список – упорядоченный (перенумерованный) набор разнотипных элементов
`set`	множество – неупорядоченный (не перенумерованный) набор разнотипных элементов
`dict`	словарь – набор разнотипных элементов, перенумерованных не числами, а уникальными значениями – ключами

Первые шесть типов данных являются *неизменяемыми*. Смысл неизменяемости можно пояснить следующим образом. Сравним работу с переменными в Pascal и Python. Когда переменная определяется в языке Pascal (в разделе определения переменных `var`), то ей выделяется область памяти и с этой областью памяти ассоциируется имя переменной. Содержимое этой области памяти может изменяться, но имя, ассоциированное с областью памяти, сохраняется.

Когда переменная определяется в языке Python (в момент первого появления в коде в присваивании или вводе с клавиатуры), то ей выделяется область памяти и с этой областью памяти ассоциируется имя переменной. Если мы хотим поменять значение неизменяемой переменной, то новому значению выделится другая область памяти, имя переменной будет перенесено на новую область памяти, а старое значение будет потеряно.

Списки, кортежи, множества и словари являются изменяемыми в том плане, что мы имеем возможность менять их содержимое, но при этом адрес переменной указанных типов не меняется (но меняются адреса их элементов).

2.1.2. Замечание по поводу оформления программного кода

Программный код, приводимый в издании, будет оформляться моноширинным шрифтом, возможно, меньшего размера, чем основной текст, и располагаться в ограничивающей рамке. Знаки операторов будут обрамляться пробелами. Ниже приведён пример кода программы и выводимого ею результата. Однострочные комментарии в программе пишутся после знака #.

Листинг 1. *Пример программного кода*

```
# суммирование матриц a и b
import numpy
```

```
a = numpy.array([
    [2, 5,-4],
    [3, 7, 2]
    ])
b = numpy.array([
    [ 0, 3, 1],
    [-5, 7,-4]
    ])
print(a+b)
```

Результат:

```
[[ 2  8 -3]
 [-2 14 -2]]
```

2.1.3. Способы задания переменных

Переменная может быть задана с помощью оператора присваивания. Ниже приведён пример задания целочисленной переменной `x`, вещественной переменной `y`, строковых переменных `z` и `w`, списка `a`, кортежа `k`, булевой переменной `b`, множества `s`, словаря `d`.

В Python поддерживается множественное присваивание. В примере ниже оно выполнено в последней строке.

Листинг 2. *Задание переменных с помощью присваивания*

```
x = 2
y = 2.1545
z = 'строка'
w = "строка"
a = [1, "d", 2.36]
k = (1, 3.14, "d")
b = True
s = {1, 2, "3"}
d = {"Иван": 28, "Пётр": 45}
alpha, beta = 30, 60
```

Переменные можно ввести с клавиатуры. Это делается с помощью функции `input`. Необязательным аргументом этой функции является строка – приглашение пользователю. По умолчанию данная функция возвращает строковую величину. Следовательно, чтобы ввести с клавиатуры целое или вещественное число, нужно результат функции `input` преобразовать с помощью функций `int` и `float` в целый и вещественный тип, соответственно. Ниже показан ввод строковой переменной `s`, целочисленной переменной `n` и вещественной переменной `f`

Листинг 3. *Ввод переменных с клавиатуры*

```
s = input("Введите строку s: ")
n = int(input("Введите целое число n: "))
f = float(input("Введите вещественное число f: "))
```

Булеву переменную более корректно задавать присваиванием. Заметим, что конструкция `b = bool(input())` будет работать. При этом, если с клавиатуры введена непустая строка символов, то в `b` будет записано `True`. Если же была введена пустая строка, то в `b` будет записано `False`.

2.1.4. Преобразование типов данных

Как уже было показано в предыдущем подпункте, строковые данные, вводимые

с клавиатуры, можно преобразовать в целочисленные и вещественные (если введённые строки можно в такие данные преобразовать). При этом применялись функции, совпадающие по имени с именами соответствующих типов данных. Такой принцип действует для всех указанных выше типов. К примеру, можно строку преобразовать в список, а список – во множество. Множество содержит только неповторяющиеся значения, и это можно использовать, например, при подсчёте уникальных элементов списка или строки.

Листинг 4. *Преобразование типов*

```
myString = "информатика"
myList = list(myString)
print(myList)
mySet = set(myList)
print(mySet)
```

Результат:

```
['и', 'н', 'ф', 'о', 'р', 'м', 'а', 'т', 'и', 'к', 'а']
{'к', 'р', 'н', 'ф', 'м', 'а', 'о', 'т', 'и'}
```

2.1.5. Допустимые имена переменных

В Python имена переменных должны подчиняться правилу: имя может состоять из букв латиницы, цифр и знака подчёркивания, причём, цифра не может быть первым символом в имени.

Желательно, чтобы имя переменной соответствовало её смысловой нагрузке в программе. Так, `value` более приемлемо, чем `v`. Вместе с тем имя не должно быть слишком длинным.

Прописные и строчные буквы в именах различаются. Так, имена `namedVertices` и `namedvertices` являются различными.

Следует избегать совпадений с уже имеющимися именами, определёнными в языке. Например, не стоит давать переменной, хранящей максимальное из двух чисел, имя `max`, так как есть функция с таким же именем.

2.1.6. Вывод текста на экран

Для вывода текстовой информации служит функция `print`, аргументами которой могут быть величины различных типов. Количество аргументов может быть произвольным.

При решении вычислительных задач будет необходимо уметь выводить вещественные значения с определённым количеством знаков после запятой. Для этого можно применять метод строки `format`.

В примере ниже показано, как во второй строке число `x` выводится в таком же виде, как оно было задано; в третьей строке число выводится в 10 позициях, из которых 3 позиции отведены на дробную часть; в четвёртой строке количество позиций на экране подбирается автоматически, из них 2 позиции отведены на дробную часть; в пятой строке число выводится в экспоненциальной форме (то есть, в виде `1.235e+02` $= 1.235 \cdot 10^2$). Заметим, что целая и дробные части вещественного числа разделяются точкой.

Листинг 5. *Форматный вывод значений*

```
x = 123.45678
print("Значение: ",x)
print("Значение: {:10.3f}".format(x))
print("Значение: {:0.2f}".format(x))
print("Значение: {:10.3e}".format(x))
```

Результат:

```
Значение:  123.45678
Значение:    123.457
Значение: 123.46
Значение:  1.235e+02
```

Иной способ вывода на экран числовых значений и результатов вычисления формул состоит в использовании f-строк. Рассмотрим пример. Пользователь вводит с клавиатуры некоторое вещественное число, а программа выводит квадрат этого числа, оставляя 2 знака после десятичной точки.

Листинг 6. *Пример использования f-строк*

```
n = float(input("Введите число: "))
print(f"Квадрат числа: {n**2 :0.2f}")
```

Результат:

```
Введите число: 3.44545
Квадрат числа: 11.87
```

Видим, что перед строкой ставится префикс `f`, а внутри строки в фигурных скобках помещено выражение, которое вычисляет квадрат введённого числа и к результату вычисления этого выражения применён форматный вывод.

2.1.7. Три полезные функции

В Python имеются три полезные функции:

Функция	Пояснение
`id(obj)`	возвращает идентификатор объекта `obj` (адрес объекта в памяти)
`type(obj)`	возвращает тип объекта `obj`
`dir(obj)`	возвращает список атрибутов объекта `obj`

Листинг 7. *Пример использования функций* `type`, `dir`, `id`

```
x = [1, 2, 4]
print(id(x))
print(type(x))
print(dir(x))
```

Результат:

```
45460136
<class 'list'>
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Можно заметить, что функция `type` выводит не просто наименование типа данных, но именует его `class`. Суть в том, что в Python самым полнейшим образом реализован объектно-ориентированный подход, и, как говорят, в Python всё является классом. Целое число – это класс, вещественное число – класс, и пр. Даже функция – это тоже класс. Поэтому далее, говоря об элементах программ, мы будем упоминать слово объект или экземпляр класса. Так как у классов определяются атрибуты (свойства и методы), то в дальнейшем мы тоже будем использовать указанные термины.

Функцию `dir` часто используют для изучения свойств и методов объектов, а также для вывода списка имён, определённых в текущей области видимости или в некотором модуле (когда аргументом функции является имя модуля).

Функция `id` полезна для понимания того, как в Python реализуется оператор присваивания. Опять же можно провести сравнение языков Pascal и Python.

Если мы в Pascal присваиваем одной переменной, например, `x`, значение другой переменной, например, `y`, то из области памяти переменной `y` в область памяти переменной `x` переносится содержимое. Имена переменных остаются закреплёнными за выделенными областями памяти.

Если мы в Python присваиваем одной переменной, например, `x`, значение другой переменной, например, `y`, то с областью памяти, в которой находится значение, ассоциированное с именем `y`, будет и ассоциировано имя `x`, так что два имени переменных будут указывать на одну и ту же область памяти. Указанный подход к понятию переменной понуждает нас более осторожно относиться к применению операций присваивания.

Листинг 8. *Что происходит при присваивании*

```
x = [1,2,3]
y = ["a","b"]
print(x,y)
print(id(x),id(y))
x = y
print(x,y)
print(id(x),id(y))
```

Результат:

```
[1, 2, 3] ['a', 'b']
45525752 45369184
['a', 'b'] ['a', 'b']
45369184 45369184
```

Так как в Python имена переменных – это «ярлыки» к областям памяти, то можно одновременно «перевешивать» эти «ярлыки« для обмена значениями переменных без использования промежуточной переменной.

Листинг 9. *Обмен значениями переменных*

```
frst = "Петя"
scnd = "Вася"
print(frst, scnd)
frst, scnd = scnd, frst
print(frst, scnd)
```

Результат:

```
Петя Вася
Вася Петя
```

2.2. Арифметические операторы

Перечислим арифметические операторы, используемые в Python.

Оператор	Пояснение
+	сложение
-	вычитание
*	умножение
/	деление
**	возведение в степень
//	неполное частное целых чисел
%	остаток от деления одного целого числа на другое

Листинг 10. *Арифметика в Python*

```
x = 17
y = 7
print('x + y =',x + y)
print('x - y =',x - y)
print('x * y =',x * y)
print('x / y =',x / y)
print('x в степени y =',x ** y)
print('a div b =',x // y)
print('a mod b =',x % y)
```

Результат:

```
x + y = 24
x - y = 10
x * y = 119
x / y = 2.4285714285714284
x в степени y = 410338673
a div b = 2
a mod b = 3
```

2.3. Определение пользовательских функций

Пользовательские функции применяются тогда, когда какой-то участок кода приходится вызывать много раз (не обязательно подряд). Приведём синтаксис определения пользовательской функции.

Определение 1. *Пользовательская функция*

```
def имя_функции(последовательность_аргументов):
    выполняемые команды
    return результат
```

Чтобы задать собственную функцию, необходимо дать ей *имя*, указать в скобках через запятые аргументы функции (скобки ставятся даже тогда, когда аргументов нет), и после двоеточия с отступом записать код функции. То, что записано далее с отступом, составляет тело функции. После `return` записывается то, что будет возвращать функция в качестве результата. Выполнив инструкцию `return`, функция заканчивает работу, так что если после указанной инструкции в теле функции есть ещё команды, они выполнены не будут.

Обязательный отступ в Python играет (почти) такую же роль, что и операторные скобки `begin..end` в Pascal. Отсюда следует вывод, что нельзя делать отступы произвольно, ибо отступы имеют значение (к примеру, в Pascal отступы не имеют

значения, но использовались для повышения удобочитаемости кода; в Python так делать нельзя). На протяжении всего кода программы нужно соблюдать размеры отступов. По умолчанию на один отступ отводят 4 пробела. Далее мы покажем, что в программе может быть несколько уровней отступов.

Определим, к примеру, функцию $f(x) = \dfrac{3x^3 - 4x^2 + x - 1}{x^3 - 11x + 1}$ и покажем, как её вызвать в программе. Заметим, что в данном примере можно обойтись без промежуточной переменной y. Мы ввели её только для того, чтобы показать, что область видимости переменных x и y ограничена функцией, и попытка вывести содержимое этой переменной на экран приводит к ошибке.

Листинг 11. *Задание и вызов функции* $f(x)$

```
def f(x):
    y = (3 * x**3 + 4 * x**2 + x + 1) / (x**3 - 11 * x + 1)
    return y

x = 10
print(f(x))
print(x)
print(y)
```

Результат:

```
3.8282828282828283
10
Traceback (most recent call last):
  File "D:\example.py", line 8, in <module>
    print(y)
NameError: name 'y' is not defined
```

2.4. Модули и пакеты

Язык Python позволяет собирать пользовательские функции в отдельные файлы с расширением py (впрочем, такое расширение имеют все файлы с исходным кодом Python). Файлы, в которых собраны определения функций и/или классов, называются *модулями*.

Несколько модулей составляют *пакет*. Физически пакет – это папка, в которую сложены файлы модулей, а также файл `__init__.py`, в котором может храниться служебная информация (данный файл может быть и пустым, но сам файл с таким именем обязан быть в папке пакета). Именем пакета является имя папки пакета.

Чтобы использовать содержимое, определённое в пакете или модуле, используется инструкция `import`. В следующем пункте мы перечислим функции, определённые в модуле `math` и далее приведём пример создания собственного пакета и его использования.

2.4.1. Модуль math

Чтобы использовать математические функции, необходимо подгрузить модуль `math`. Перечислим функции, определённые в этом модуле.

Функция	Пояснение
`fabs(x)`	абсолютная величина (модуль) числа x
`floor(x)`	целая часть «снизу», наибольшее целое число, меньшее или равное x
`ceil(x)`	целая часть «сверху», наименьшее целое число, большее или равное x

`exp(x)`	экспонента e^x
`log(x)`	натуральный логарифм $\ln x$
`log(x,a)`	логарифм по произвольному основанию $\log_a x$
`pow(x,n)`	степень x^n
`sqrt(x)`	квадратный корень $\sqrt{x}$
`pi`	число $\pi \approx 3.14159$
`e`	число $e \approx 2.71828$
`sin(x)`	синус $\sin x$
`cos(x)`	косинус $\cos x$
`tan(x)`	тангенс $\mathrm{tg}\, x$
`asin(x)`	арксинус $\arcsin x$
`acos(x)`	арккосинус $\arccos x$
`atan(x)`	арктангенс $\mathrm{arctg}\, x$
`radians(x)`	перевод угла x из градусной меры в радианную
`degrees(x)`	перевод угла x из радианной меры в градусную

Как получить доступ к данным функциям? Если мы запишем инструкцию `import math`, то получим доступ ко всем функциям и объектам, определённым в модуле. Вызов функций в тексте программы осуществляется с помощью селектора (имени модуля, после которого идет точка, а за ней следует имя функции). В примере ниже показано вычисление значения выражения $\sin \frac{\pi}{4}$.

Листинг 12. *Использование математического модуля*

```
import math
print(math.sin(math.pi/4))
```

Результат:

```
0.7071067811865475
```

Инструкция `import` позволяет задать псевдоним загружаемому модулю (это делается, к примеру, если имя модуля слишком длинно). Тогда в тексте программы вместо исходного имени модуля нужно использовать псевдоним.

Листинг 13. *Использование псевдонима*

```
import math as m
print(m.sin(m.pi/4))
```

Чтобы получить доступ только к некоторой функции, определённой в модуле, можно использовать инструкцию

`from math import` *`имя_функции`*

В этом случае в тексте программы селектор не используется.

Листинг 14. *Загрузка отдельных функций*

```
from math import sin, pi
print(sin(pi/4))
```

Заметим, что селектор также не используется в случае, когда загрузка содержимого модуля была осуществлена с использованием инструкции `from math import *`

Листинг 15. *Загрузка всех функций*

```
from math import *
print(sin(pi/4))
```

2.4.2. Создание собственного модуля

Создадим модуль, в котором определим функции котангенса и арккотангенса в виде: $\operatorname{arcctg} x = \frac{\pi}{2} - \operatorname{arctg} x$ и $\operatorname{ctg} x = \frac{\cos x}{\sin x}$.

Дадим название модулю `mymodule`. Поместим модуль в пакет с именем `mypack`. В текущей папке создадим папку с именем `mypack`, поместим в неё пустой файл `__init__.py` и файл `mymodule.py` со следующим содержимым.

Листинг 16. *Содержимое модуля* `mymodule.py`

```
from math import sin, cos, atan, pi
def cot(x):
    return cos(x)/sin(x)
def acot(x):
    return pi / 2 - atan(x)
```

Проверим работу пакета. В текущей папке создадим файл `test.py` со следующим содержимым (вычислим значения числовых выражений $\operatorname{ctg} \frac{\pi}{4}$ и $\operatorname{arcctg} 1$).

Листинг 17. *Содержимое файла* `test.py`

```
from math import *
from mypack import mymodule as mym
print(mym.cot(pi/4))
print(mym.acot(1))
```

Результат:

```
1.0000000000000002
0.7853981633974483
```

2.5. Инструкция ветвления if

Инструкция ветвления применяется в том случае, если необходимо выполнить те или иные действия в зависимости от истинности или ложности некоторого условия (или условий).

Определение 2. *Синтаксис ветвления*

```
if условие_1:
    команды, выполняемые, если условие_1 истинно
elif условие_2:
    команды, выполняемые, если условие_2 истинно
..
else:
    команды, выполняемые, если все условия ложны
```

Поясним синтаксис. Сначала проверяется истинность *условия*$_1$. Если оно истинно, то выполняются все действия, записанные после двоеточия с отступом.

Если *условие*$_1$ ложно, то проверяется истинность *условия*$_2$, записанного после `elif`, что является сокращением пары `else if`. Если *условие*$_2$ истинно, то выполняются все дальнейшие действия, записанные с отступом после двоеточия. Блок `elif` может повторяться в условной инструкции произвольное количество раз.

Если ни одно из проверяемых условий не оказалось истинным, то выполняются действия, записанные с отступом после `else`.

2.5.1. Условия

Условия – это выражения, которые возвращают логическое значение `True` или `False` (более научно – логические выражения).

Простейшие условия можно сформировать с помощью операторов сравнения:

Оператор	Пояснение
==	равно
!=	не равно
>	больше
>=	больше или равно
<	меньше
<=	меньше или равно

Если мы желаем узнать, находится ли какое-либо значение в строке, списке, кортеже, множестве или словаре, то используем оператор членства `in`. Приведём пример такого рода проверки.

Листинг 18. *Проверка членства*

```
print("Иван" in "Иван Иванович Иванов")
print("Иван" in {"Иван": 28,"Пётр": 45})
print("Иван" in ["Степан", "Пётр"])
print("Иван" in {"Иван", "Пётр", "Степан"})
print("Маня" in ("Иван", "Пётр", "Степан"))
```

Результат:

```
True
True
False
True
False
```

Если мы желаем узнать, указывают ли два имени переменных на одну и ту же область памяти, нужно использовать оператор тождественности `is`. Приведём пример такого рода проверки.

Листинг 19. *Проверка тождественности*

```
x = 2
y = x
z = 2
print(x is y)
print(x is z)
print(id(x), id(y), id(z))
```

Результат:

```
True
True
1878877344 1878877344 1878877344
```

Условия можно соединять с помощью логических операторов `or` (*или*, дизъюнкция), `and` (*и*, конъюнкция), `not` (*не*, отрицание). Кроме того, в Python поддерживаются множественные (двойные, тройные и т. д.) неравенства. К примеру, условие $x \in [-3; 5)$ можно записать в виде `x >= -3 and x < 5`, либо в виде `-3 <= x < 5`.

Напомним, что дизъюнкция нескольких условий применяется в том случае, когда нам нужно, чтобы хотя бы одно из условий выполнилось (дизъюнкция ложна тогда и только тогда, когда все условия ложны, а если хотя бы одно условие истинно, то и дизъюнкция истинна). Конъюнкция нескольких условий применяется в том случае, когда нам нужно, чтобы все условия выполнились (конъюнкция истинна тогда и только тогда, когда хотя бы одно из условий истинно, а если хотя бы одно условие ложное, то и конъюнкция ложна). Отрицание истинного условия есть

ложное условие, и наоборот, отрицание ложного условия есть истинное условие.

Указанные свойства логических операций можно систематизировать в так называемой таблице истинности.

A	B	not A	A or B	A and B
False	False	True	False	False
False	True	True	True	False
True	False	False	True	False
True	True	False	True	True

Заметим, что операторы or и and могут быть применены не только к двум операндам, но и к большему их количеству. К примеру, могут быть конструкции A or B or C or D или A and B and C and D.

Если в логическом выражении нет скобок, то наивысший приоритет имеет оператор not, затем – оператор and, и самый низкий приоритет у or.

Приведём пример решения задачи проверки, находится ли число y между числами x и z (то есть, выполняется ли одно из условий – $x \leqslant y \leqslant z$ или $z \leqslant y \leqslant x$). Условия, прописанные в примере, не содержат скобок, так как учитывается приоритет выполнения логических операторов (операторы сравнения имеют более высокий приоритет, чем логические операторы).

Листинг 20. *Приоритет логических операторов*

```
x = float(input("Введите x: "))
y = float(input("Введите y: "))
z = float(input("Введите z: "))
# способ 1
if x <= y and y <= z or z <= y and y <= x:
    print("y находится между x и z")
else:
    print("y не находится между x и z")
# способ 2
if x <= y <= z or z <= y <= x:
    print("y находится между x и z")
else:
    print("y не находится между x и z")
```

Результат:

```
Введите x: 0.2
Введите y: 1.8
Введите z: 0.7
y не находится между x и z
y не находится между x и z
```

2.5.2. Инструкция try .. except

Нередки случаи, когда функция не может быть выполнена от каки-то конкретных наборов аргументов. К примеру, при попытке преобразовать в число строку, не содержащую символов числа, возникает ошибка.

Листинг 21. *Ошибка при выполнении функции*

```
print(int("Петя"))
```

Результат:

```
Traceback (most recent call last):
  File "D:\exampleerror.py", line 1, in <module>
    print(int("Петя"))
ValueError: invalid literal for int() with base 10: 'Петя'
```

Если важно, чтобы программа не прерывала работу по причине ошибки, то блок кода, в котором могут возникнуть непредвиденные ошибки, нужно заключать в инструкцию `try .. except`.

Определение 3. *Инструкция* `try .. except`

```
try:
    команды, которые могут вызвать исключение
except тип исключения:
    команды, выполняемые в случае, если возникло исключение
```

Имеется множество видов исключений. Приведём несколько примеров:

- `FloatingPointError` – ошибка при работе с вещественными числами,
- `OverflowError` – результат арифметической операции слишком велик,
- `ZeroDivisionError` – ошибка деления на ноль,
- `NameError` – отсутствует переменная с указанным именем,
- `ValueError` – функция получает необрабатываемый аргумент,
- `TypeError` – функция применена к объекту несоответствующего типа.

Если мы не знаем, какой вид исключения указать после `except`, можно указать исключение общего типа `Exception`. Так, предыдущий пример можно записать следующим образом.

Листинг 22. *Ошибка при выполнении функции*

```
try:
    print(int("Петя"))
except Exception:
    print("Преобразование не удалось")
```

Результат:

```
Преобразование не удалось
```

Приведём также более полный синтаксис инструкции `try`.

Определение 4. *Инструкция* `try .. except .. else .. finally`

```
try:
    команды, которые могут вызвать исключение
except тип исключения1:
    команды, выполняемые в случае, если возникло исключение1
except тип исключения2:
    команды, выполняемые в случае, если возникло исключение2
..
else:
    команды, выполняемые в случае, если исключений не было
finally:
    команды, необходимые для окончания обработки данных
```

2.6. Инструкции циклов

Циклы применяются в том случае, если несколько раз подряд нужно выполнить одни и те же действия.

2.6.1. Цикл for

Цикл с параметром `for` применяется в том случае, когда количество повторений (итераций) нам известно заранее.

Определение 5. *Синтаксис цикла* `for`

```
for переменная in последовательность:
    команды тела цикла
else:
    команды, выполняемые после окончания цикла
```

Цикл перебирает элементы *последовательности*, присваивая их значения *переменной*. Каждое такое присваивание соответствует одной итерации цикла. На каждой итерации выполняются команды, записанные с отступом после двоеточия. Необязательная часть `else` может быть выполнена после окончания перебора элементов *последовательности*.

В качестве *последовательности* можно взять список, кортеж, словарь, объект `range` и множество (последнее – только в случае, когда не важно, в каком порядке будут перебраны элементы).

Отдельно рассмотрим использование `range`. Пример ниже показывает особенности этой функции.

Листинг 23. *Функция* `range`

```
n = 5
print(range(n))
print(type(range(n)))
print(list(range(n)))
```

Результат:

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

Видно, что команда `print` не выводит содержимое объекта `range`. После применения преобразования к типу `list` мы видим последовательность целых чисел, начинающаяся с 0 и заканчивающаяся числом $n-1$. Следующий пример показывает шире возможности `range`.

Листинг 24. *Функция* `range`

```
n = 10
print(list(range(n)))
print(list(range(3,n)))
print(list(range(2,n,2)))
print(list(range(n,1,-2)))
print(list(range(n-20,n+20,4)))
```

Результат:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 4, 5, 6, 7, 8, 9]
[2, 4, 6, 8]
[10, 8, 6, 4, 2]
[-10, -6, -2, 2, 6, 10, 14, 18, 22, 26]
```

Приведём пример использования цикла с параметром и функцией `range`. Найдём конечную сумму $\sum\limits_{n=1}^{100} \frac{1}{n^2}$. Заметим, что для улучшения точности необходимо начать с самых маленьких слагаемых (то есть, с $n = 100$).

Листинг 25. *Вычисление конечной суммы*

```
mySum = 0
for n in range(100,0,-1):
    mySum = mySum + 1 / (n * n)
print(mySum)
```

Результат:

```
1.634983900184893
```

2.6.2. Цикл while

Цикл с условием `while` применяется в том случае, когда количество повторений (итераций) нам заранее не известно.

Определение 6. *Синтаксис цикла* `while`

```
while условие:
    команды тела цикла
else:
    команды, выполняемые после окончания цикла
```

Перед выполнением очередной итерации проверяется *условие*, и если оно истинно, то итерация выполняется (выполняются *команды тела цикла*), а если *условие* ложно, то цикл завершается. Если была указана необязательная часть `else`, то выполняются команды этой необязательной части.

Приведём пример использования цикла с условием для вычисления конечной суммы $\sum_{n=1}^{100} \frac{1}{n^2}$.

Листинг 26. *Вычисление конечной суммы*

```
mySum = 0
n = 100
while n>0:
    mySum = mySum + 1 / (n * n)
    n = n - 1
print(mySum)
```

Результат:

```
1.634983900184893
```

Заметим, что изменение переменной в результате арифметической операции можно записывать более кратко:

Длинная запись	Короткая запись
`n = n + c`	`n += c`
`n = n - c`	`n -= c`
`n = n * c`	`n *= c`
`n = n / c`	`n /= c`
`n = n // c`	`n //= c`
`n = n % c`	`n %= c`
`n = n ** c`	`n **= c`

2.6.3. Операторы break и continue

Оператор `break` используется для того, чтобы досрочно прервать выполнение цикла. В последующем пункте будет показан пример использования данного оператора.

Оператор `continue` начинает новую итерацию цикла, минуя команды, записанные после этого оператора в теле цикла.

2.6.4. Вычисление приближённого значения нуля функции методом пассивного поиска

Известно, что функция $f(x) = x^3 - 4x + 5$ обращается в ноль в единственной точке отрезка $[a; b] = [-3; -2]$. Вычислить приближённое значение нуля этой функции методом пассивного поиска с точностью $\varepsilon = 0.0001$.

Если $\hat{x}$ – точное значение, а $\tilde{x}$ – приближённое значение нуля функции $f(x)$, удовлетворяющее заданной точности, то это означает выполнение неравенства $|\tilde{x} - \hat{x}| < \varepsilon$.

Разделим отрезок $[-3; -2]$ на n равных частей точками:

$$x_0 = -3, \quad x_1, \quad x_2, \quad \ldots, \quad x_{i-1}, \quad x_i, \quad \ldots, \quad x_n = -2,$$

Если на концах некоторого частичного отрезка $[x_{i-1}; x_i]$ $(i = 1, \ldots, n)$ функция $f(x)$ принимает разные знаки, то данный отрезок содержит ноль функции. В качестве приближенного значения нуля функции можно принять середину частичного отрезка:

$$\tilde{x} = \frac{x_{i-1} + x_i}{2}.$$

Чтобы истинное значение нуля было удалено от приближённого менее, чем на ε, нужно, чтобы половина длины частичного отрезка была меньше ε, а полная длина частичного отрезка была меньше 2ε:

$$x_i - x_{i-1} < 2\varepsilon \quad \text{или} \quad \frac{x_n - x_0}{n} < 2\varepsilon.$$

Отсюда следует, что количество частичных отрезков n необходимо вычислять по формуле

$$n = \left\lceil \frac{x_n - x_0}{2\varepsilon} \right\rceil,$$

где $\lceil x \rceil$ – целая часть x «сверху», то есть, наименьшее целое число, большее или равное x. Тогда длина частичного отрезка

$$d = \frac{x_n - x_0}{n},$$

формула i-й точки деления

$$x_i = x_0 + i \cdot d, \qquad i = 0, \ldots, n.$$

Листинг 27. *Метод пассивного поиска нуля функции*

```
import math
x_o = -3
x_n = -2
eps = 0.0001
n = math.ceil((x_n - x_o) / (2 * eps))
d = (x_n - x_o) / n
```

```
def f(x):
    return x ** 3 - 4 * x + 5

for i in range(n):
    x_pred = x_o + i * d
    x_succ = x_o + (i + 1) * d
    if f(x_pred) * f(x_succ) < 0:
        break

print(f"Найден отрезок: [{x_pred}; {x_succ}]")
print(f"Приближённое значение нуля: {(x_pred + x_succ) / 2 :0.4f}")
```

Результат:

```
Найден отрезок: [-2.4568, -2.4566]
Приближённое значение нуля функции: -2.4567
```

2.7. Списки

2.7.1. Задание списков

Список представляет собой упорядоченную последовательность разнотипных элементов, которые, в свою очередь, также могут быть списками. Одномерные списки могут быть использованы в качестве представления векторов, двумерные – матриц.

Обратиться к `i`-му элементу списка `a` можно по номеру: `a[i]`. Нумерация элементов начинается с нуля. Если список включает в себя другие списки (вложенные списки), то индексы элементов перечисляются в квадратных скобках: `a[i][j]`.

Для определения длины списка *a* применяется функция `len(a)`.

Как уже было показано выше, список можно задать с помощью присваивания. Также можно ввести элементы списка с клавиатуры, используя при этом цикл. К примеру, зададим трёхэлементный список вещественных чисел.

Листинг 28. *Ввод элементов списка с клавиатуры*

```
myList = [ ]
for i in range(3):
    myList.append(float(input(f"Введи {i}-й элемент списка: ")))
print(myList)
```

Результат:

```
Введи 0-й элемент списка: 2
Введи 1-й элемент списка: 5
Введи 2-й элемент списка: 4
[2.0, 5.0, 4.0]
```

Одномерный список можно внести с клавиатуры целиком, ставя между элементами списка символ-*разделитель*. Идея такова: после ввода строки с клавиатуры, необходимо её разрезать на элементы, используя метод строки `split(`*`разделитель`*`)`. После этого мы получим список строк. Если требуется получить не список строк, а, например, список чисел, то каждый элемент этого списка нужно преобразовать с помощью функций преобразования типов `int` или `float`. Распределить эти функции по списку (то есть, применить их к каждому элементу списка как к аргументу) нужно с помощью функции `map(`*`функция, список`*`)`. Результат, возвращаемый функцией `map`, имеет тип `<class 'map'>`, поэтому к нему придётся применить функцию преобразования в список `list`.

Листинг 29. *Ввод списка строкой*

```
myString = input('Введите числа через запятые: ')
myList = myString.split(',')
print(list(map(float,myList)))
```

Результат:

```
Введите числа через запятые: 1, 2.3, 5.66
[1.0, 2.3, 5.66]
```

Наконец, список можно получить с помощью функции `list` из строки, кортежа, словаря или множества.

Листинг 30. *Получение списка цифр числа и преобразованием словаря*

```
a = list(str(12346))
print(a)
b = list({"x": 1, "y": 2})
print(b)
```

Результат:

```
['1', '2', '3', '4', '6']
['x', 'y']
```

2.7.2. Срезы

Можно извлекать элементы из списка не по отдельности, но группой, используя срезы вида `a[i:j:n]`, где `a` – список, `i` – номер начального элемента среза, `j` – номер конечного элемента среза, `n` – шаг, с которым выбираются элементы списка. Заметим, что элемент с номером `j` на экран не выводится. Кроме того, можно использовать отрицательные номера элементов: последний элемент списка имеет номер -1, предпоследний -2 и т. д.

Листинг 31. *Срезы*

```
myList = ["a", "b", "c", "d", "e"]
print("myList[1:3] =",myList[1:3])
print("myList[1:] =",myList[1:])
print("myList[:3] =",myList[:3])
print("myList[-2:] =",myList[-2:])
print("myList[:-2] =",myList[:-2])
print("myList[:] =",myList[:])
print("myList[::2] =",myList[::2])
print("myList[:4:2] =",myList[:4:2])
```

Результат:

```
myList[1:3] = ['b', 'c']
myList[1:] = ['b', 'c', 'd', 'e']
a myList[:3] = ['a', 'b', 'c']
myList[-2:] = ['d', 'e']
myList[:-2] = ['a', 'b', 'c']
myList[:] = ['a', 'b', 'c', 'd', 'e']
myList[::2] = ['a', 'c', 'e']
myList[:4:2] = ['a', 'c']
```

При извлечении среза получается новый список со своим идентификатором. Данный эффект полезен, если необходимо сделать копию списка, независимую от своего оригинала. К сожалению, элементы списков будут иметь одинаковые иден-

тификаторы.
Листинг 32. *Результат получения среза*

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

Результат:

```
45787776 1945068704
48301960 1945068704
```

2.7.3. Lambda-функции

Если тело функции можно записать в одну-две строчки кода, либо вызов функции производится в единственном месте основного кода, то можно не задавать такую функцию с помощью `def`, а определить в виде lambda-функции:
Определение 7. *Синтаксис lambda-функции*

```
lambda последовательность_аргументов : выражение
```

Функция вычисляет некоторое *выражение* и возвращает его значение в качестве результата. Чтобы применить lambda-функцию к конкретным аргументам, её определение окружают круглыми скобками и рядом в круглых скобках записывают эти аргументы.

Покажем, как можно написать функцию, возвращающую `True`, если целое число является точным квадратом, и `False`, если целое число точным квадратом не является.
Листинг 33. *Пример использования lambda-функции*

```
from math import sqrt, floor
print((lambda n : floor(sqrt(n)) == sqrt(n))(100))
```

Результат:

```
True
```

2.7.4. Полезные функции, применяемые к спискам

Функция `map(f,a)`, имея аргументами заголовок `f` функции одного аргумента `f(x)` и список `a`, имеющий вид `[a[0],a[1],...,a[n]]`, возвращает выражение вида `[f(a[0]),f(a[1]),...,f(a[n])]`. Другими словами, она распределяет некоторую функцию `f` по элементам списка, делая их аргументом функции, или *векторизует* функцию `f`. Напомним, что результат, возвращаемый функцией `map`, имеет тип `<class 'map'>`, поэтому к нему придётся применить функцию преобразования в список `list`.

Заметим, что удобно в качестве функции `f` брать lambda-функцию.

Приведём пример, в котором покажем, как можно проверить, являются ли элементы целочисленного списка точными квадратами.
Листинг 34. *Проверка элементов списка*

```
from math import sqrt, floor
myList = [1, 4, 5, 9, 10, 12]
myResult = list(map(lambda n : floor(sqrt(n)) == sqrt(n),myList))
print(myResult)
```

Результат:

```
[True, True, False, True, False, False]
```

Функция `all(a)`, применённая к списку `a`, возвращает `True`, если все элементы списка имеют значение `True`, либо не являются нулями (могут быть ненулевыми числами, непустыми строками, непустыми списками и т. п.). Так, можно изменить результат, выводимый в предыдущем примере, если нам нужно узнать, *все* ли элементы списка являются точными квадратами:

Листинг 35. *Проверка элементов списка*

```
from math import sqrt, floor
myList = [1, 4, 5, 9, 10, 12]
myResult = all(list(map(lambda n : floor(sqrt(n)) == sqrt(n),myList)))
print(myResult)
```

Результат:

```
False
```

Приведём ещё несколько полезных функций и методов списков.

Функция	Описание
`min(a)`	Вычисляет минимальный элемент последовательности `a`
`max(a)`	Вычисляет максимальный элемент последовательности `a`
`sorted(a)`	Сортирует последовательность `a`
`a.sort()`	Сортирует список `a`
`a.append(e)`	Добавляет в конец списка `a` элемент `e`
`a.insert(n,e)`	Добавляет в список `a` элемент `e` в позицию с номером `n`
`a.remove(e)`	Удаляет из списка `a` первое вхождение элемента `e`
`a.pop(n)`	Удаляет из списка `a` элемент в позиции с номером `n`
`a.count(e)`	Возвращает количество вхождений элемента `e` в список `a`

Заметим, что функция `sorted` возвращает новый список, а метод `list.sort` меняет исходный список. Кроме того, три первые функции применимы не только к спискам, но к любым последовательностям (кортежи, словари), а первые две функции применимы и к последовательностям отдельных значений. Так, верны записи `min(1, 2)` и `min([1, 2])`. Функция `count` также применима к кортежам и строкам.

Приведём пример, в котором отсортируем список, содержащий данные вида `["имя", рост]` по имени, а затем по убыванию роста. По умолчанию сортировка осуществляется по первому элементу каждой «строки» списка.

Листинг 36. *Сортировка*

```
myList = [["Маша", 167], ["Ваня", 172], ["Петя", 181]]
print(sorted(myList))
myList.sort()
print(myList)

myList = [["Маша", 167], ["Ваня", 172], ["Петя", 181]]
```

```
print(sorted(myList, key = lambda elem : elem[0]))
myList.sort(key = lambda elem : elem[0])
print(myList)

myList = [["Маша", 167], ["Ваня", 172], ["Петя", 181]]
print(sorted(myList, key = lambda elem : elem[1], reverse = True))
myList.sort(key = lambda elem : elem[1], reverse = True)
print(myList)
```

Результат:

```
[['Ваня', 172], ['Маша', 167], ['Петя', 181]]
[['Ваня', 172], ['Маша', 167], ['Петя', 181]]
[['Ваня', 172], ['Маша', 167], ['Петя', 181]]
[['Ваня', 172], ['Маша', 167], ['Петя', 181]]
[['Петя', 181], ['Ваня', 172], ['Маша', 167]]
[['Петя', 181], ['Ваня', 172], ['Маша', 167]]
```

2.7.5. Генератор списков (списочное включение)

В Python имеется конструкция, называемая *генератором списка* или *списочным включением*, которая позволяет сделать код короче при автоматическом создании списка (в Python есть отдельное понятие *генератора*; термины «генератор» и «генератор списка» не стоит смешивать).

Определение 8. *Синтаксис генератора линейного списка*

```
[выражение for элемент in последовательность if условие ]
```

Элементы списка создаются по формуле, записанной в *выражении*, пока цикл перебирает элементы из некоторой *последовательности*, которая может быть объектом range, списком, множеством, кортежем или словарём, и для каждого элемента проверяется *условие*, только при выполнении которого результат *выражения* добавляется в список.

Чтобы создать многомерный список, нужно в качестве *выражения* использовать генератор списка.

Листинг 37. *Примеры применения списочных включений*

```
print([n for n in range(5)])
print([i + j for j in range(2) for i in range(3)])
print([[i + j for j in range(2)] for i in range(3)])
a = [[1, 2, 3], [4, 5, 6]]
print([[elem[n]**2 for n in range(len(elem))] for elem in a])
```

Результат:

```
[0, 1, 2, 3, 4]
[0, 1, 2, 1, 2, 3]
[[0, 1], [1, 2], [2, 3]]
[[1, 4, 9], [16, 25, 36]]
```

Решим задачу: создать список случайных целых чисел и вывести все номера позиций минимальных элементов списка.

Листинг 38. *Решение задачи с помощью списочного включения*

```
from random import randint
a = [randint(-2,3) for i in range(10)]
b = [i for i in range(10) if a[i] == min(a)]
print(f"Исходный список: {a}.\nПозиции минимальных элементов: {b}.")
```

Результат:

```
Исходный список: [-2, 2, -2, -1, -2, 2, -1, 2, 3, -2].
Позиции минимальных элементов: [0, 2, 4, 9].
```