Тема 2. ОСНОВЫ ЯЗЫКА PYTHON

Оглавление

Тема 2. ОСНОВЫ ЯЗЫКА *PYTHON*

Любой язык программирования определяется совокупностью трех составляющих: алфавитом, синтаксисом и семантикой.

Алфавит – фиксированный для данного языка набор символов (букв, цифр, специальных знаков и т.д.), которые могут быть использованы при написании программы.

Синтаксис – правила построения из символов алфавита специальных конструкций, с помощью которых составляется алгоритм.

Семантика – система правил толкования конструкций языка.

Таким образом, программа составляется с помощью соединения символов алфавита в соответствии с синтаксическими правилами и с учетом правил семантики.

2.1 Алфавит языка

В основе всех конструкций языка программирования лежит алфавит. Допустимый набор символов языка *Python* включает:

– латинские буквы: *A, B*, …, *Z, a, b*, …, *z*;

– цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;

– специальные знаки: +, –, /, *, (,), [,], <, >, =, :, а также точка, запятая, нижнее подчеркивание, одинарные и двойные кавычки.

Кроме того, в языке *Python* существует ряд последовательностей из представленных специальных знаков, рассматриваемых в качестве неделимых элементов (составных символов). К ним относят:

– обозначения используемых в программировании альтернатив математических знаков нестрогого неравенства: >= и <=;

– используемые для экранирования и определения многострочных литералов (текста) утроенные одинарные или двойные кавычки: """…""" или '''…'''.

Символы алфавита используются для создания **лексем** (токенов) – минимальных единиц языка, имеющих самостоятельный смысл. Лексемы делятся на две большие группы: *служебные слова* (ключевые или зарезервированные) и *идентификаторы*. В

первую группу входят предопределенные смысловые элементы, имеющие отношение к конструкциям базовых операторов языка. Ко второй группе относят обозначения объектов (переменных, функций, классов и т. д.), создаваемых программистом в процессе написания кода.

Для работы со служебными словами в *Python* существует модуль *keyword*. С его помощью можно как вывести полный список ключевых слов, актуальных для используемой версии языка, так и проверить, является ли выбранное имя идентификатора ключевым словом:

```
import keyword
# получение общего списка ключевых слов
print(keyword.kwlist)
# проверка принадлежности некоторой строки
# к списку ключевых слов
print(keyword.iskeyword('class'),
      keyword.iskeyword('lesson'))
```

Результат:

```
['False', 'None', 'True', 'and', 'as', 'assert',
'async', 'await', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally',
'for', 'from', 'global', 'if', 'import', 'in', 'is',
'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise',
'return', 'try', 'while', 'with', 'yield']
True False
```

Некоторые ключевые слова могут стать доступными после подключения модуля *__future__*, обеспечивающего обратную совместимость при появлении новых версий языка.

2.2 Переменные, константы и литералы

Совокупность величин, с которыми работает компьютер, принято называть данными. По отношению к программе данные делятся на исходные, получаемые в процессе вычислений промежуточные и результаты (окончательные данные). В программировании оперируют понятиями переменная и константа.

Переменная – область для хранения данных. В *Python* нет команды для объявления переменной.

Имена переменных используются для доступа к данным. Данные в *Python* представлены в форме объектов, каждый из ко-

торых имеет свой тип, например, *int* (целое число), *str* (строка) и др. Следует отметить, переменные хранят не сам объект, а ссылку на объект, то есть адрес объекта в памяти компьютера.

В программе связь между данными и переменными устанавливается с помощью оператора присваивания, обозначаемого знаком равенства. Выполняется оператор стандартным образом: сначала вычисляется выражения справа от знака равенства, а затем полученное значение записывается в переменную, указанную слева от знака равенства.

Переменная создается в момент первого присваивания ей значения, например:

```
title = 'Учебное пособие'
number = 10
```

Переменной *title* присвоено значение 'Учебное пособие', а в *number* хранится значение 10.

В ходе выполнения программы переменная может неоднократно изменять свое значение:

```
message = 'Здравствуйте!'
print(message)
message = 'До свидания!'
print(message)
```

Здесь переменной *message* сначала присвоено значение 'Здравствуйте!' с последующим выводом его на экран, а затем с помощью повторного присваивания оно было изменено и также выведено на экран.

Имена переменных подчиняются следующим правилам:

1. Название должно начинаться с буквы или символа подчеркивания, но не с цифры.

2. Имя переменной не должно содержать пробелы. Если имя переменной должно состоят из нескольких слов, они должны быть разделены знаком нижнего подчеркивания.

3. В качестве идентификаторов запрещено использовать зарезервированные ключевые слова языка.

4. Следует выбирать осмысленные имена, говорящие о назначении данных, на которые они ссылаются.

Следует отметить, *Python* является чувствительным к регистру: *count* и *Count* – разные переменные. Например:

```
count = 100
```

```
Count = 1000
print(count) # будет выведено 100
print(Count) # будет выведено 1000
```

В большинстве языков программирования **константа** – тип переменной, значение которой нельзя изменить на протяжении всего жизненного цикла программы.

Однако в *Python* нет встроенного механизма для объявления констант: общепринятый подход основан на соглашении рекомендательного характера об именовании. Имена констант должны быть написаны прописными буквами с подчеркиваниями между словами.

В дополнение к соглашению об именовании константы обычно определяются в отдельном модуле с его последующим подключением к основному файлу программы.

Файл *constant.py*:

```
# определяем константы
PI = 3.14
GRAVITY = 9.8
```

Файл *main.py*:

```
# подключаем файл с константами
import constant
print(constant.PI) # выведет 3.14
print(constant.GRAVITY) # выведет 9.8
```

Создан файл *constant.py* с двумя глобальными переменные *PI* и *GRAVITY*, значения которых не должны изменяться в ходе дальнейшего вычислительного процесса. После этого создан файл *main.py*, код которого содержит подключение (импортирование) ранее созданного модуля с константами. Затем значения констант выводятся на экран.

Литералы – представления фиксированных значений в программе. Это могут быть числа, символы, строки и т. д., например, 'ЮРГПУ(НПИ)', 7, 12.0, '*D*'.

Литералы часто используются для присваивания значений переменным или константам. Например:

```
title = 'ЮРГПУ (НПИ)'
```

В приведенной строке *title* – переменная, а 'ЮРГПУ (НПИ)' – литерал.

Присутствующие в коде программы фиксированные значения можно разделить на четыре вида: числовые, логические, строковые/символьные и специальные литералы.

Числовые литералы являются неизменяемыми на протяжение всего вычислительного процесса. В табл. 2.1 представлены примеры числовых литералов.

Таблица 2.1 – Числовые литералы

Тип	Пример	Примечание
Десятичный	6, 11, -57	Обычные числа
Бинарный	0*b*102, 0*b*12	Начинается с 0*b*
Восьмеричный	0*o*14	Начинается с 0*o*
Шестнадцатеричный	0*x*14	Начинается с 0*x*
Литерал типа с плавающей точкой	11.6, 3.14	Содержит плавающие десятичные точки
Сложный литерал	7 + *j*8	Числовые литералы в форме *a* + *jb*, где *a* – действительная часть, а *b* – мнимая

Логические литералы представляют собой служебные слова *True* и *False*.

Строковые литералы представляют собой последовательности символов, заключенные в одинарные или двойные кавычки. Например:

```
some_string = 'Hello World!
```

В приведенной строке '*Hello World*!' – строковый литерал, присвоенный переменной *some_string*.

Символьные литералы – *Unicode*-символы, заключенные в кавычки. Например:

```
some_character = 'S'
```

В приведенной строке '*S*' – символьный литерал, который присвоен переменной *some_character*.

Python содержит **специальный литерал** *None*, используемый для указания null-переменной. Например:

```
value = None
print(value)
```

В результате выполнения приведенного кода на экран будет выведено *None*, так как переменной *value* не присвоено значение.

2.3 Типы данных

Встроенные типы данных языка *Python* делят на две большие группы: ***примитивные типы*** и ***структуры данных***.

К примитивным типам относят числа, логические значения и строки, к структурам данных – списки, кортежи, множества и словари. Числа в *Python* делятся на: целые, вещественные и комплексные.

2.3.1 Целые числа

Целые числа могут быть сколько угодно большими (длинная арифметика), ограничение на диапазон их значений накладывает система, в которой работает программист. Ключевое слово, зарезервированное для обозначения целого типа данных, ***int***.

Ранее, в *Python* 2.*x*, существовало разделение целых чисел по хранимому диапазону значений, и присутствовал идентификатор *long*, используемый для работы с длинными целыми числами.

Атрибут *int_info* модуля *sys* позволяет получить информацию (только для чтения) о точности и внутреннем представлении типа:

```
import sys
print(sys.int_info)
```

В табл. 2.2 представлены компоненты атрибута и их значения, полученные после выполнения приведенного фрагмента кода.

Таблица 2.2 – Описание атрибута *int_info*

Компонент	Значение	Описание
bits_per_digit	30	Количество битов, содержащихся в каждой цифре. В *Python* целые числа хранятся во внутреннем представлении по основанию 2***int_info.bits_per_digit*
sizeof_digit	4	Размер в байтах типа языка *C*

По умолчанию стандартные числа расцениваются как числа в десятичной системе. Для работы с другими системами счисления, используемыми в компьютерной технике, используются префик-

сы: 0*b* – для двоичной, 0*o* – для восьмеричной и 0*x* – шестнадцатеричной.

Стоит отметить, при передаче числа в функцию *print* для вывода на консоль, *по умолчанию*, не зависимо от указанного префикса, оно будет выводиться в десятичной системе:

```
a = 0b11
print(a)    # 3 в десятичной системе
b = 0o11
print(b)    # 9 в десятичной системе
c = 0x11
print(c)    # 17 в десятичной системе
```

Для печати числа в требуемой системе счисления необходимо использовать встроенные функции конвертации:

```
print(c, bin(c), oct(c), hex(c))
```

В результате выполнения приведенного кода на экран будут выведены через пробел следующие значения: 17, 0*b*10001, 0*o*21 и 0*x*11.

Функция *bin*() выполняет конвертацию в двоичную систему счисления, *oct*() – в восьмеричную и *hex*() – в шестнадцатеричную систему счисления. Преобразуемое значение может быть связано с идентификатором, как в рассмотренном случае, а может быть задано в виде числового литерала.

2.3.2 Вещественные числа

Тип ***float*** в *Python* соответствует числам двойной точности стандарта *IEEE*-754, регулирующего представление вещественных чисел на аппаратном уровне.

Атрибут *float_info* модуля *sys* позволяет получить информацию о точности и внутреннем представлении типа:

```
import sys
print(sys.float_info)
```

В табл. 2.3 представлено описание атрибута *float_info*.

Для хранения данных типа *float* отводится 64 бита (разряда). Данный формат позволяет представлять числа с плавающей точкой с высокой точностью и диапазоном значений.

Таблица 2.3 – Описание атрибута *float_info*

Компонент	Значение	Описание
max	1.7976931348623157*e*+308	Максимально представимое положительное конечное число с плавающей точкой
max_exp	1024	Максимальное целое число *e* такое, что *radix***(*e*-1) является представимым конечным числом с плавающей точкой
max_10_exp	308	Максимальное целое число *e* такое, что 10***e* находится в диапазоне представимых конечных чисел с плавающей точкой
min	2.2250738585072014*e*-308	Минимально представимое положительное нормированное число с плавающей точкой
min_exp	-1021	Минимальное целое число *e* такое, что *radix***(*e*-1) является нормализованным числом с плавающей точкой
min_10_exp	-307	Максимальное целое число *e* такое, что 10***e* находится в диапазоне представимых конечных чисел с плавающей точкой
dig	15	Максимальное количество десятичных цифр, которое может быть достоверно представлено в формате с плавающей точкой
mant_dig	53	Точность с плавающей точкой: число цифр от базового радиуса в значении и с плавающей точкой
epsilon	2.220446049250313*e*-16	Разница между 1.0 и наименьшим значением больше 1.0, которое представляется как

Компонент	Значение	Описание
		число с плавающей запятой
radix	2	Основание представления степени
rounds	1	Целочисленная константа, представляющая режим округления, используемый для арифметических операций. Это отражает значение системного макроса *FLT_ROUNDS* во время запуска интерпретатора

Число с плавающей запятой состоит из набора отдельных двоичных разрядов, условно разделенных на знак, порядок и мантиссу (рис. 2.1).

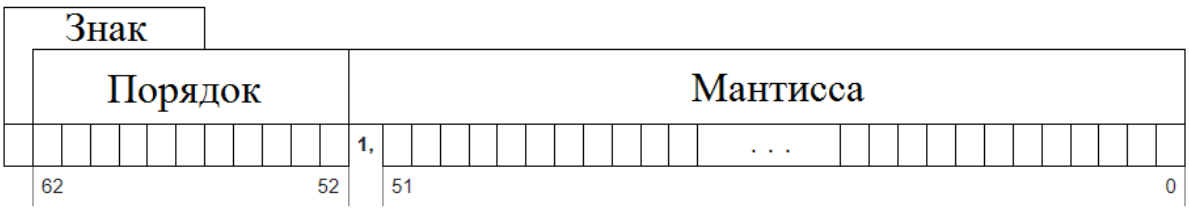

Рис. 2.1. Схема представления вещественного числа двойной точности

Знак – один бит, указывающий знак всего числа с плавающей точкой (0 для положительных чисел, 1 – отрицательных). Порядок (экспонента, показатель степени) записывается как целое число в коде со сдвигом, а мантисса – в нормализованном виде, своей дробной частью в двоичной системе счисления. Так как старший двоичный разряд мантиссы вещественного числа в нормализованном виде всегда равен «1», то его не записывают.

Порядок – бинарное представление целого числа, в которое нужно возвести 10, чтобы при перемножении на мантиссу в нормализованном виде получить исходное число. В стандарте *IEEE*-754 дополнительно введено смещение, упрощающее операции сравнения.

Множество вещественных чисел, точно представимых в памяти компьютера в форме с плавающей точкой, является ограни-

ченным и дискретным. Дискретность является следствием ограниченного числа разрядов мантиссы.

Вычисления с вещественными числами компьютер выполняет приближенно. Погрешность таких вычислений называют погрешностью машинных округлений:

```
print(0.1 + 0.1 + 0.1 == 0.3)
print(0.15 + 0.05 + 0.1 == 0.3)
print(0.15 + 0.15 == 0.3)
```

Результат:

```
False
False
True
```

Следует отметить, допустимые вещественные числа на числовой прямой распределены неравномерно: плотность размещения уменьшается по мере приближения к граничным значениям диапазона значений.

2.3.3 Комплексные числа

Тип данных ***complex*** относится к категории неизменяемых и хранит пару значений типа *float*, одно из которых представляет действительную часть комплексного числа, а другое – мнимую. Мнимый литерал можно задать с помощью числа с плавающей точкой или десятичного дробного числа с фиксированной точностью добавлением в конец буквы *j*, обозначающей квадратный корень из –1.

Литерал комплексного числа в полной форме записывается путем объединения обеих частей знаком "+" или "–":

```
complexNumber = 1.5+2j
print(complexNumber)   # (1.5+2j)
```

Тип *complex* в языке *Python* представлен классом *complex*():

```
x = complex(0.44, 0.58)
print(x)   # (0.44+0.58j)
print(x.real)  # 0.44
print(x.imag)  # 0.58
```

Атрибуты *x.real* и *x.imag* обеспечивают доступ к обеим частям комплексного объекта, они доступны только для чтения.

Данный класс позволяет:

– создать комплексное число со значением переданных аргументов: действительной и мнимой частью;

– преобразовать строку с записью комплексного числа в комплексное число.

Для работы с рассматриваемым типом чисел используется модуль *cmath* с комплексными версиями большинства тригонометрических и логарифмических функций.

2.3.4 Логические значения

Логический тип ***bool*** представлен двумя постоянными значениями *False* и *True*. Значения используются для представления истинности.

Если преобразовать логическое *True* к типу *int*, то получится 1, а преобразование *False* даст 0. При обратном преобразовании число 0 преобразуется в *False*, а любое ненулевое число в *True*. При преобразовании *str* в *bool* пустая строка преобразовывается в *False*, а любая непустая строка в *True*.

2.3.5 Строки

Для обозначения типа данных зарезервировано ключевое слово ***str***, текстовые данные обрабатываются с помощью встроенного класса *str*(). В *Python* 3.*x* строки представляют неизменяемые последовательности символов в кодировке *Unicode*, заключенные в одинарные или двойные кавычки, например "*hello*" и '*hello*':

```
title = "ЮРГПУ (НПИ) имени М.И. Платова"
print(title)      # ЮРГПУ (НПИ) имени М.И. Платова
city = 'Новочеркасск'
print(city)   # Новочеркасск
```

Если в строке много символов, ее можно разместить на разных строках кода частями, создав *multi-line* строку с помощью утроенных одинарных или двойных кавычек:

```
title = """ЮРГПУ (НПИ) имени М.И. Платова -
первый вуз юга России"""
```

Устроенные кавычки можно использовать в качестве многострочного комментария:

```
'''
Это комментарий
'''
```

или

```
"""
Это комментарий
"""
```

При использовании тройных одинарных кавычек не стоит путать их с комментариями: если текст в тройных одинарных кавычках присваивается переменной, то это строка, а не комментарий.

Строка может содержать ряд специальных символов – управляющих последовательностей:

\\ экранирование символа '\';

\' экранирование одинарной кавычки;

\" экранирование двойной кавычки;

a звуковой сигнал или предупреждение;

b возврат на одну позицию;

n перенос строки на новую;

r возврат каретки;

t горизонтальный *Tab*;

v вертикальный *Tab*;

ooo восьмеричное значение;

xHH шестнадцатеричное значение.

Пример использования управляющих последовательностей:

```
text = "Сообщение:\n\"Экзамен сдан\""
print(text)
```

Консольный вывод программы:

```
Сообщение:
"Экзамен сдан"
```

Язык *Python* позволяет встраивать в строку значения других переменных. Для этого переменные размещаются в фигурных скобках {}, а перед всей строкой ставится символ *f*:

```
userName = "Иван"
userAge = 25
user = f"Имя: {userName} - возраст: {userAge}"
print(user)   # Имя: Иван - возраст: 25
```

В данном случае вместо {*userName*} и {*userAge*} будут подставлены значения переменных *userName* и *userAge*, соответственно.

Получить доступ к отдельным символам в строке можно двумя способами: по индексу и с помощью среза.

Доступ по индексу выполняется с использование квадратных скобок. Индекс всегда должен быть целым, не обязательно положительным числом:

```
city = 'Новочеркасск'
print(city[0])      # Н
print(city[2])      # в
print(city[-1])     # к
print(city[-4])     # а
```

Нумерация символов строки слева направо начинается с 0, справа на лево с -1 .

При попытке получить доступ к символу с индексом, превышающим длину строки, будет выдана ошибка *IndexError*.

Срез используется для выделения части строки. Он состоит из индекса и диапазона. Базовая структура среза выглядит следующим образом:

```
Строка [начальный_символ : конечный_символ + 1]
```

В табл. 2.4 представлены результаты применения различных вариантов среза к строке.

Таблица 2.4 – Получение срезов строки *example* = 'абвгдежзик'

Фрагмент кода	Результат	Описание
`example[2:5]`	вгд	Символы с индексом 2, 3, 4
`example[:5]`	абвгд	Первые пять символов
`example[5:]`	ежзик	Символы, начиная с индекса 5 и до конца
`example [-2:]`	ик	Последние два символа
`example [:]`	абвгдежзик	Вся строка
`example [1:7:2]`	бге	Со второго по шестой символы, через один
`example [:: -1]`	кизжедгвба	Обратный шаг, строка наоборот

Отсутствие значение индекса начала или конца будет по умолчанию приравнивать начальную или конечную позицию среза нулю. Если же использовать отрицательные индексы срез будет формироваться с конца строки.

Работа со строковым типом данных поддерживается обширным встроенным функционалов. В табл. 2.5 приведен список наиболее часто используемых методов работы со строками, для ознакомления с полным списком следует воспользоваться документацией самого языка. Вызов метода осуществляется с помощью конструкции `строка.метод()`.

Таблица 2.5 – Методы обработки строк

Название метода	Описание
`split(символ)`	Разбивает строки по заданному разделителю (пробелом по умолчанию)
`replace(шаблон, замена)`	Замена шаблона в строке
`find(подстрока, начало, конец)`	Поиск подстроки в строке. Возвращает индекс первого вхождения слева. Если подстроки в строке нет, возвращает -1
`index(подстрока, начало, конец)`	Поиск подстроки в строке. Возвращает индекс первого вхождения. Если подстроки в строке нет, возвращает *ValueError*
`isdigit()`	Проверяет, состоит ли строка из цифр. Возвращает *True* или *False*
`isalpha()`	Проверяет, состоит ли строка из букв. Возвращает *True* или *False*
`islower()`	Проверяет, состоит ли строка из символов в нижнем регистре. Возвращает *True* или *False*
`isupper()`	Проверяет, состоит ли строка из символов в верхнем регистре. Возвращает *True* или *False*
`upper()`	Преобразует строку к верхнему регистру
`lower()`	Преобразует строку к нижнему регистру
`ord(символ)`	Преобразует символ в *ASCII*-код
`chr(число)`	Преобразует *ASCII*-код в символ

Наряду со специальными методами для обработки строк применяются и стандартные операторы *Python*, речь о которых

пойдет в пособии далее. Например, объединение строк выполняет оператор '+, а перебор символов в строке осуществим с помощью циклов.

2.3.6 Динамическая типизация

Python является языком со строгой динамической типизацией.

Строгая или сильная типизация запрещает производить неявные преобразования типов. Например, следующая инструкция завершится ошибкой:

```
some_number = 1 + '1'
```

Результат выполнения:

```
unsupported operand type(s) for +: 'int' and 'str'
```

Динамическая составляющая типизации позволяет определять тип переменной исходя из присваиваемого ей значения. Так, при присвоении строки в двойных или одинарных кавычках переменная имеет тип *str*, целое число автоматически определяет тип переменной как *int*, а объект типа *float* появляется в случае присваивания числа, разделенного точкой на целую и дробную части.

В процессе работы программы можно изменить тип переменной, то присвоив ей значение другого вида. С помощью встроенной функции *type*() можно узнать текущий тип переменной:

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

Динамическая типизация позволяет максимально естественно абстрагироваться от типов и заниматься обобщенным программированием, однако является причиной низкой скорости выполнения кода.

Для языков программирования с динамической типизацией применяется концепция утиной типизация, согласно которой важен не конкретный тип или класс объекта, а поддерживаемые объектом свойства и методы:

```
print(len('123'))      # 3
print(len([1, 2]))     # 2
print(len(123))        # TypeError
```

В приведенных строках кода выполняется попытка определить длину объекта и вывести на экран это значение. Функция len(), не проверяя тип объекта, обращается к методу __len__(). Если он существует, то функция возвращает длину, если метода нет – будет выдана ошибка.

2.3.7 Преобразование типов

В программировании преобразование типа (англ. *type conversion, typecasting*) – процесс приведения значения одного типа в значение другого типа.

В силу строгой типизации *Python* не поддерживает неявные преобразования, за исключением работы с числами. Оба операнда в арифметических операциях должны представлять один и тот же тип, если это не так, интерпретатор пытается автоматически выполнить преобразования в соответствии со следующими правилами:

1. Если один из операндов операции представляет комплексное число (тип *complex*), то другой операнд также преобразуется к типу *complex*.

2. Если один из операндов представляет тип *float*, то второй операнд также преобразуется к типу *float*.

При неявном преобразовании типов не допускается потери данных:

```
num1 = 305
num2 = 3.05
num = num1 + num2
print("Тип данных в num1:", type(num1))
print("Тип данных в num2:", type(num2))
print("Значение num:", num)
print("Тип данных в num:", type(num))
```

Результат:

```
Тип данных в num1: <class 'int'>
Тип данных в num2: <class 'float'>
Значение num: 308.05
Тип данных в num: <class 'float'>
```

Значение переменной *num* определяется в виде результата сложения *num*1 и *num*2. Интерпретатор *Python* всегда преобразу-

ет меньший по диапазону тип в больший, чтобы избежать потери данных, поэтому тип *num* определен как *float*.

В явном преобразовании программист сам заменяет текущий тип данных объекта на требуемый, используя встроенные функции, такие как *int*(), *float*(), *str*():

```
num1 = "305"
num2 = 3.05
print("Тип данных в num1:", type(num1))
print("Тип данных в num2:", type(num2))
num1 = float(num1)
print("Тип данных в num1:", type(num1))
num = num1 + num2
print("Значение num:", num)
print("Тип данных в num:", type(num))
```

Результат:

```
Тип данных в num1: <class 'str'>
Тип данных в num2: <class 'float'>
Тип данных в num1: <class 'float'>
Значение num: 308.05
Тип данных в num: <class 'float'>
```

Переменная *num*1 преобразована в тип *float*, переменная *num* создана с типом *float*.

Явное преобразование может завершиться ошибкой, если значение преобразуемой переменной не может соответствовать новому типу:

```
x = "десять"
x = int(x)    # ValueError
```

При попытке преобразовать *x* из строки в целое число будет выдано сообщение о недопустимом литерале для представления числа в десятичной системе счисления.

2.4 Ввод и вывод данных

2.4.1 Работа с консолью

Ввод данных с клавиатуры осуществляется с помощью функции *input*(), возвращающей для считанной строки значение, которое сразу можно присвоить переменной. По умолчанию возвра-

щается текстовая строка. Для получения значений других типов, нужно либо выполнить явное преобразование сразу после считывания, либо совместить считывание с преобразованием:

```
a = input()       # считывание строки
a = int(a)        # преобразование в целое число
x = int(input())  # считывание и преобразование
```

Если в функцию *input*() поместить строковый литерал, на экран будет выведено сообщение, предшествующее считыванию данных:

```
y = float(input('y = ')) # вывод приглашения ко вводу
```

Для вывода значений переменных и выражений на консоль используется функция *print*(). Также при помощи этой функции можно выводить значения нескольких выражений, перечислив их через запятую, или воспользоваться *f*-строками:

```
print(5 + 10, 6 - 3)
a = float(input('Введите a: '))
print(a)
print('Как вас зовут?')
name = input()
print(f'Здравствуйте, {name}!')
```

Данные и результат:

```
15 3
Введите a: 4.5
4.5
Как вас зовут?
Иван
Здравствуйте, Иван!
```

В первой строке выводятся результаты арифметических операций, во второй и третьей строках показан результат вывода на экран ранее введенных и обработанных данных. Строки с четвертой по шестую демонстрируют диалог с пользователем, приветствие формируется с помощью *f*-строки.

Функция *print*() по умолчанию выводит передаваемое ей значение на новой строке. Изменить это поведение можно с помощью параметра *end*, который определяет символы, добавляемые в конце выводимой строки.

```
print('Python', end = ' - ')
print('язык', end = ' ')
print('программирования')
```

Результат работы всех трех функций *print*() будет отображаться на одной строке, а после слова '*Python*' помимо пробела будет добавлен '-'.

Аргумент функции *print*() может быть сформирован с помощью метода *format*(), осуществляющего подставку значений в шаблон строки:

```
print('{}, {}, {}'.format('A', 'B', 'C')) #A, B, C
print('{1}, {0}, {2}'.format('A', 'B', 'C')) #B, A, C
```

В строке места вставки (поля замены) определяются фигурными скобками. Внутри скобок могут указываться индексы или ключи аргументов, переданных в метод *format*().

Данный метод позволяет задавать ширину поля и выравнивание:

```
u = {'name': 'Антон','age': 20}
print('{name:10}-{age:4}-'.format(**u))
print('{name:>10}-{age:>4}-'.format(**u))
print('{name:^10}-{age:^4}-'.format(**u))
```

Результат вывода:

```
Антон     -  20-
     Антон-  20-
  Антон   - 20 -
```

Для вещественных чисел можно задавать количество отображаемых знаков после десятичной точки:

```
print('{}'.format(4/3))
print('{:.2f}'.format(4/3))
print('{0:10.2f}'.format(4/3))
print('{0:e}'.format(4/3))
```

Результат вывода:

```
1.3333333333333333
1.33
      1.33
1.333333e+00
```

Кроме того, форматирование аргумента функции *print*() может быть выполнено с помощью оператора '%'. Данный способ заимствован из функции *printf*() языка *C*:

```
a = 10 / 6
print('%.2f' % a) # 1.67
```

Следует отметить, оператор форматирования строк выполняет округление вещественных чисел, а не простое отбрасывание «лишних» цифр.

2.4.2 Работа с файлами

Файл – набор данных, сохраненный в виде последовательности битов на компьютере.

Язык *Python* работает с двумя типами файлов: текстовыми и бинарными.

В текстовых файлах хранятся последовательности символов, которые понимает человек. Блокнот и другие стандартные редакторы умеют читать и редактировать этот тип файлов. В бинарных файлах данные отображаются в закодированной форме (с использованием только нулей и единиц вместо простых символов). Работа с этим типом файлов несколько сложнее. Нередко их обрабатывают с помощью специальных модулей (*pickle*, *struct*).

В этом разделе будет рассмотрена работа с файлами с использованием встроенного функционала.

Любую операцию с файлом можно разделить на три этапа:

– открытие файла;

– выполнение операции (запись, чтение);

– закрытие файла.

Для открытия файла используется функция *open()*, которая возвращает файловый объект и имеет следующий синтаксис:

```
f = open(file_name, access_mode),
```

где *file_name* – имя открываемого файла. Если файл находится в директории проекта, путь к нему указывать не нужно; *access_mode* – режим открытия файла.

В табл. 2.6 приведен полный список режимов открытия файла (текстовых и бинарных).

Таблица 2.6 – Режимы открытия файла

Название режима	Описание
r	Только для чтения
w	Только для записи. Создаст новый файл, если не найдет с указанным именем
rb	Только для чтения (бинарный)
wb	Только для записи (бинарный). Создаст новый файл, если не найдет с указанным именем
r+	Для чтения и записи
rb+	Для чтения и записи (бинарный)
w+	Для чтения и записи. Создаст новый файл для записи, если не найдет с указанным именем
wb+	Для чтения и записи (бинарный). Создаст новый файл для записи, если не найдет с указанным именем
a	Откроет для добавления нового содержимого. Создаст новый файл для записи, если не найдет с указанным именем
a+	Откроет для добавления нового содержимого. Создаст новый файл для чтения записи, если не найдет с указанным именем
ab	Откроет для добавления нового содержимого (бинарный). Создаст новый файл для записи, если не найдет с указанным именем
ab+	Откроет для добавления нового содержимого (бинарный). Создаст новый файл для чтения записи, если не найдет с указанным именем

Функция *read*() используется для чтения содержимого файла после открытия его в режиме чтения *r*.

Синтаксис функции имеет вид:

```
file.read(size),
```

где *file* – объект файла; *size* – количество символов, которые нужно прочитать. Если количество символов не указано, то файл будет прочтен целиком.

Функция *readline*() используется для построчного чтения содержимого файла. Она используется для крупных файлов. С ее помощью можно получать доступ к любой строке в любой момент. Ее синтаксис аналогичен функции *read*().

Функция *write*() используется для записи в файлы Python, открытые в режиме записи *w*.

Синтаксис функции имеет вид

```
file.write(string),
```

где *string* – строка, которую необходимо записать.

Функция *rename*() используется для переименования файлов. Для ее использования нужно импортировать модуль ***os***:

```
import os
os.rename(src,dest),
```

где *src* – старое имя файла; *dest* – новое имя файла.

В *Python* возможно узнать текущую позицию в файле с помощью функции *tell*(). Таким же образом можно изменить текущую позицию командой *seek*().

После завершения работы с файлом следует его закрыть с помощью *close*(). Свойство файлового объекта *closed* позволяет проверить закрыт ли файл.

В табл. 2.7 представлены методы работы с файловыми объектами.

Таблица 2.7 – Режимы открытия файла

Название метода	Описание
file.close()	Закрывает открытый файл
file.fileno()	Возвращает целочисленный дескриптор файла
file.flush()	Очищает внутренний буфер
file.isatty()	Возвращает *True*, если файл привязан к терминалу
file.next()	Возвращает следующую строку файла
file.read(*n*)	Чтение первых *n* символов файла
file.readline()	Читает одну строчку строки или файла
file.readlines()	Читает и возвращает список всех строк в файле
file.seek(*offset*[,*whene*])	Устанавливает текущую позицию в файле
file.seekable()	Проверяет, поддерживает ли файл случайный доступ. Возвращает *True* или *False*
file.tell()	Возвращает текущую позицию в файле
file.truncate(*n*)	Уменьшает размер файл. Если *n* указала, то файл обрезается до *n* байт, если параметр не задан – до текущей позиции
file.write(*str*)	Добавляет строку *str* в файл
file.writelines(*sequence*)	Добавляет последовательность строк в файл

Пример. Запросить у пользователя его имя и название файла со значениями целочисленных переменных. Результаты вычисления суммы чисел, а также имя пользователя сохранить в новый файл:

```
print('Как вас зовут?')
name = input()
print(f'Введите название файла, {name}.')
title = input()

f = open(title,'r')   # открытие файла для чтения
a = int(f.readline()) # считывание значения
b = int(f.readline()) # считывание значения
f.close()             # закрытие файла

result = a + b

f = open('result.txt','w')# открытие файл для записи
f.writelines(f'''Пользователь: {name}
Результат вычислений: {result}''')
f.close()             # закрытие файла
```

Хорошей практикой при работе с файлами является применение оператора *with*:

```
with open("test.txt", "r") as f:
# работа с файлом
```

По окончании работы с файлом, операция его закрытия и освобождения занятой памяти будет выполнена автоматически.

2.5 Базовые операторы

Операторы выполняют операции над определенными значениями, называемыми операндами. *Python* делит операторы на следующие группы:

– арифметические операторы;
– операторы присваивания;
– операторы сравнения;
– логические операторы;
– операторы тождественности;
– операторы принадлежности;
– поразрядные (битовые) операторы.

Для обозначения большинства операторов используются один или несколько символов, логические операторы обозначаются зарезервированными словами.

2.5.1 Арифметические операторы

Все арифметические операторы являются бинарными, другими словами, для выполнения операции требуются два операнда (аргумента оператора). В отличие от других языков программирования *Python* обладает расширенным набором базовых арифметических операций, перечень которых представлен в табл. 2.8.

Таблица 2.8 – Арифметические операторы в языке *Python*

Оператор	Использование в коде	Описание
+	*a* + *b*	Возвращает сумму операндов
-	*a* - *b*	Возвращает разность операндов
*	*a* * *b*	Возвращает произведение операндов
/	*a* / *b*	Возвращает частное операндов
%	*a* % *b*	Возвращает остаток от деления левого операнда на правый. Если левый операнд меньше правого, то результатом будет левый операнд
**	*a* ** *b*	Выполняет возведение операнда *a* в степень *b*
//	*a* // *b*	Возвращает целую часть частного операндов

Следует отметить, комплексные числа поддерживают не все арифметические операции – недоступны нахождение целой части и остатка от деления.

2.5.2 Операторы присваивания

В *Python* операции присваивания делятся на три вида: простое и составное присваивание, а также выражения присваивания.

Простое присваивание значения переменной имеет следующий синтаксис:

```
var = value
object.attr_var = value
iterator[x] = value
```

```
iterator[start:stop:step] = value
```

Во время выполнения операции присваивания значения переменной сначала вычисляется выражение *value*, а затем связывается полученное значение с целевой ссылкой. Данное связывание не зависит от типа значения. *Python* не проводит строгого различия между вызываемыми и невызываемыми объектами, как это принято делать в некоторых языках программирования, поэтому допускается связывать с переменными функции, методы, типы и другие вызываемые объекты точно так же, как числа, строки, списки и т. д. Это обусловлено тем, что функции и им подобные объекты являются объектами первого класса.

Детали связывания зависят от вида целевой ссылки. В качестве целевой ссылки в операции присвоения может выступать идентификатор, ссылка на атрибут, а также ссылка на индексированный элемент или срез.

При ***групповом присваивании*** выражение, находящееся в правой части, вычисляется только один раз, независимо от количества целевых ссылок, указанных в инструкции. Каждая из целевых ссылок, в порядке следования слева направо, связывается с единственным объектом, аналогично нескольким поочередным операциям присваивания:

```
a = b = c = 0
print(a, b, c)  # 0 0 0
a = b = c = 10 + 8
print(a, b, c) # 18 18 18
```

Составное присваивание, например, *count += value* не может создать новую ссылку и выполняет только повторное связывание.

Перечень составных операторов присваивания представлен в табл. 2.9.

Таблица 2.9 – Составные операторы присваивания в языке *Python*

Оператор	Использование в коде	Альтернативный вариант	Описание
+=	*a += b*	*a = a + b*	Увеличивает правый операнд на значение левого и присваивает полученную сумму левому операнду
-=	*a -= b*	*a = a - b*	Уменьшает правый операнд на значение левого и присваивает полученную разность

Оператор	Использование в коде	Альтернативный вариант	Описание
			левому операнду
*=	*a* *= *b*	*a* = *a* * *b*	Умножает правый операнд на левый и присваивает результат левому операнду
/=	*a* / =*b*	*a* = *a* / *b*	Делит правый операнд на левый и присваивает результат левому операнду
%=	*a* % = *b*	*a* = *a* % *b*	Вычисляет остаток от деления левого операнда на правый и присваивает результат левому операнду
**=	*a* ** = *b*	*a* = *a* ** *b*	Возводит левый операнд в степень правого и присваивает результат левому операнду
//=	*a* //= *b*	*a* = *a* // *b*	Производит целочисленное деление левого операнда на правый и присваивает результат левому операнду
&=	*a* &= *b*	*a* = *a* & *b*	Выполняет поразрядное логическое умножение левого и правого операндов и присваивает результаты левому операнду
|=	*a* |= *b*	*a* = *a* | *b*	Выполняет поразрядное логическое сложение левого и правого операндов и присваивает результаты левому операнду
^=	*a* ^= *b*	*a* = *a* ^ *b*	Выполняет поразрядное логическое сложение по модулю 2 левого и правого операндов и присваивает результаты левому операнду
<<=	*a* <<= *b*	*a* = *a* << *b*	Выполняет сдвиг влево для левого операнда на количество битов, определенное правым операндом и присваивает результат левому операнду
>>=	*a* >>= *b*	*a* = *a* >> *b*	Выполняет сдвиг вправо для левого операнда на количе-

Оператор	Использование в коде	Альтернативный вариант	Описание
			ство битов, определенное правым операндом и присваивает результат левому операнду

Следует отметить, в *Python* нет традиционных операторов инкремента и декремента, таких как ++ или --. Вместо них используются составные операторы, которые объединяют оператор присваивания = с математической операцией сложения += или вычитания -=.

Базовые побитовые операции будут рассмотрены более подробно в одном из следующих разделов.

Выражения присваивания появились в *Python* 3.8. Это способ присваивания значения переменной в выражении с использованием обозначения *num* := *value*. Данный оператор позволяет как присваивать значение переменной, так и возвращать это значение в одном и том же выражении. Например, в коде программы необходимо присвоить переменной *num* значение 15, и затем вывести ее значение. Эти действия можно выполнить в одной инструкции, используя выражения присваивания:

```
print(num := 15) # 15
```

Если сделать то же самое с помощью обычного оператора присваивания, будет сгенерирована ошибка, поскольку *num* = 15 ничего не возвращает.

2.5.3 Операторы сравнения

Для создания условий используются операторы сравнения (табл. 2.10). Они возвращают результат в виде булевых значений *True* или *False* в случае истинности или ложности условия, соответственно.

Таблица 2.10 – Операторы сравнения в языке *Python*

Оператор	Использование в коде	Описание
==	*a* == *b*	Проверяет равенство значений операндов. Если они равны – условие является истиной
!=	*a* != *b*	Проверяет неравенство значений

Оператор	Использование в коде	Описание
		операндов. Если они не равны – условие является истиной
>	*a* > *b*	Проверяет значение левого операнда. Если оно больше значения правого – условие является истиной
<	*a* < *b*	Проверяет значение левого операнда. Если оно меньше значение правого – условие является истиной
>=	*a* >= *b*	Проверяет значение левого операнда. Если оно больше, либо равно значению правого – условие является истиной
<=	*a* <= *b*	Проверяет значение левого операнда. Если оно меньше, либо равно значению правого – условие является истиной

2.5.4 Логические операторы

Логические операторы используются для объединения нескольких операторов сравнения (табл. 2.11).

Таблица 2.11 – Логические операторы в языке *Python*

Оператор	Использование в коде	Описание
and	*x* > *y and z* == 10	Логическое И (умножение), возвращает значение *True* если оба утверждения верны
or	*x* > 5 *or x* > 10	Логическое ИЛИ (сложение), возвращает *True* если одно из утверждений верно
not	*not*(*x* > 5 *or x* < *y*)	Логическое НЕ (отрицание или инверсия), возвращает *False* если результат *True* и наоборот

2.5.5 Операторы тождественности

Операторы тождественности проверяют, находятся ли два значения (или две переменные) по одному адресу в памяти (табл. 2.12).

Таблица 2.12 – Операторы тождественности в языке *Python*

Оператор	Использование в коде	Описание
is	*x is not y*	Возвращает *True*, если переменные указывают на один объект
is not	*x is y*	Возвращает *True*, если переменные указывают на разные объекты

Рассмотрим следующие инструкции кода:

```
x1 = 10
y1 = 10
x2 = 'Привет'
y2 = 'Привет'
x3 = [1,3,5]
y3 = [1,3,5]
print(x1 is not y1) # Вывод: False
print(x2 is y2) # Вывод: True
print(x3 is y3) # Вывод: False
```

В приведенном фрагменте кода *x*1 и *y*1 – целочисленные переменные с одинаковыми значениями, поэтому они равны и идентичны, следовательно, при использовании оператора *is not* будет выведен результат *False*. Переменные *x*2 и *y*2 являются строками с одинаковыми значениями, поэтому объекты идентичны. Переменные *x*3 и *y*3 – списки. Они равны, но не идентичны, так как размещаются интерпретатором в разных участках в памяти.

2.5.6 Операторы принадлежности

Операторы принадлежности проверяют наличие значения или переменной в последовательности (строке, списке, кортеже, множестве или словаре). Другими словами, проверяют вхождение элемента в коллекцию (табл. 2.13).

Строка представляет набор символов, и с помощью оператора *in* можно проверить, есть ли в ней некоторая подстрока:

```
x = 'Операторы принадлежности '
print('О' in x) # Вывод: True
print('Принадлежности' not in x) # Вывод: True
```

Символ `'О'` есть в *x*, а подстрока `'Принадлежности'` в *x* отсутствует (*Python* чувствителен к регистру).

Таблица 2.13 – Операторы принадлежности в языке *Python*

Оператор	Использование в коде	Описание
in	‘П’ *in x*	Возвращает *True*, если значение или переменная есть в последовательности
not in	1 *not in x*	Возвращает *True*, если значения или переменной нет в последовательности

В словаре можно проверить только присутствие ключа, а не значения:

```
y = {1:'a',2:'b'}
print(1 in y) # Вывод: True
print('b' in y) # Вывод: False
```

Переменная *y* представляет словарь, следовательно, 1 и 2 – ключи, а '*a*' и `'b'` – значения в словаре, поэтому результатом проверки наличия символа `'b'` в словаре является *False*.

2.5.7 Поразрядные (битовые) операторы

Поразрядные операторы инициируют выполнение операций над отдельными разрядами или битами чисел на аппаратном уровне (табл. 2.14).

Таблица 2.14 – Поразрядные (битовые) операторы в языке *Python*

Оператор	Использование в коде	Описание
&	*a* & *b*	Поразрядное логическое умножение операндов *a* и *b*
|	*a* | *b*	Поразрядное логическое сложение операндов *a* и *b*
^	*a* ^ *b*	Поразрядное логическое сложение по модулю 2 (исключающее ИЛИ) операндов *a* и *b*
~	~*a*	Положительные числа преобразуются в отрицательные со сдвигом на единицу, и наоборот. Выражение ~*a* аналогично -(*a*+1).
<<	*a* << *b*	Битовый сдвиг влево для операнда *a* на количество бит *b*
>>	*a* >> *b*	Битовый сдвиг вправо для операнда *a* на количество бит *b*

Данный факт позволяет значительно увеличить производительность и оптимизировать код программы. Поразрядные операции производятся только над целыми числами и используются для работы с флагами и битовыми масками, сжатия данных, управления отдельными битами регистров процессора и внешних устройств.

Рассмотрим несколько примеров, иллюстрирующих поразрядные операции.

Поразрядное логическое умножение используется для выключения битов. Любой бит операнда, установленный в 0, вызывает установку соответствующего бита результата также в 0.

Пример 1. Выполнить поразрядное логическое умножение для чисел 10 и 12.

Исходные данные: → & 1010 → Результат:
$10_{10} = 1010_2$ 1100 $1000_2 = 8_{10}$
$12_{10} = 1100_2$ 1000

Код *Python*:

```
x = 10
y = 12
res = x & y
print(x, "&", y, "=", res)
print(bin(x), "&", bin(y), "=", bin(res))
```

Результат:

```
10 & 12 = 8
0b1010 & 0b1100 = 0b1000
```

Поразрядное логическое сложение используется для включения битов. Любой бит операнда, установленный в 1, вызывает установку соответствующего бита результата также в 1.

Пример 2. Выполнить поразрядное логическое сложение для чисел 13 и 17.

Исходные данные: → | 1101 → Результат:
$13_{10} = 1101_2$ 10001 $11101_2 = 29_{10}$
$17_{10} = 10001_2$ 11101

Код *Python*:

```
x = 13
y = 17
res = x | y
```

```
print(x, "|", y, "=", res)
print(bin(x), "|", bin(y), "=", bin(res))
```

Результат:

```
13 | 17 = 29
0b1101 | 0b10001 = 0b11101
```

Поразрядное логическое сложение по модулю 2 (исключающее ИЛИ) устанавливает значение бита результата в 1, если значения в соответствующих битах операндов различны.

Пример 3. Выполнить поразрядное логическое сложение для чисел 19 и 27.

Исходные данные:	$\rightarrow$			$\rightarrow$	Результат:
$19_{10} = 10011_2$		^	10011		
$27_{10} = 11011_2$			11011		$1000_2 = 8_{10}$
			01000		

Код *Python*:

```
x = 19
y = 27
res = x ^ y
print(x, "^", y, "=", res)
print(bin(x), "^", bin(y), "=", bin(res))
```

Результат:

```
19 ^ 27 = 8
0b10011 ^ 0b11011 = 0b1000
```

Следует отметить, при выводе информации на консоль нули в старших разрядах числа не отображаются.

Сдвиги влево и вправо можно использовать вместо непосредственного умножения или деления первого операнда на удвоенное значение второго операнда:

```
x = 16
res = x << 2
print(x, "<< 2 =", res)
print(bin(x), "<< 2 =", bin(res))
res = x >> 2
print(x, ">> 2 =", res)
print(bin(x), ">> 2 =", bin(res))
```

Результат:

```
16 << 2 = 64
0b10000 << 2 = 0b1000000
16 >> 2 = 4
0b10000 >> 2 = 0b100
```

Число 16_{10} в двоичной системе счисления равно 10000_2, при сдвиге влево на 2 разряда будет получено число 1000000_2, равное 64_{10}, а при сдвиге вправо на 2 разряда – 100_2, равное 4_{10}.

2.5.8 Приоритет операторов

Последовательность выполнения операций в сложном выражении определяется их приоритетом, а в случае равенства приоритетов у подряд идущих операций – «местоположением» в выражении. Вычисления производятся слева направо, то есть, если в выражении встретятся операторы одинаковых приоритетов, первым будет выполнен тот, что слева. Оператор возведения в степень исключение из этого правила. Из двух операторов **** сначала выполнится правый, а потом левый. В табл. 2.15 перечислены операторы языка *Python* в порядке убывания их приоритета.

Таблица 2.15 – Поразрядные (битовые) операторы в языке *Python*

Оператор	Описание
()	Скобки
**	Возведение в степень
+*x*, -*x*, ~*x*	Унарные плюс, минус и битовое отрицание
*, /, //, %	Умножение, деление, целочисленное деление, остаток от деления
+, -	Сложение и вычитание
<<, >>	Битовые сдвиги
&	Поразрядное логическое умножение
^	Поразрядное логическое сложение по модулю 2
|	Поразрядное логическое сложение
==, !=, >, >=, <, <=, *is, is not, in, not in*	Сравнение, проверка идентичности, проверка вхождения
not	Логическое отрицание
and	Логическое умножение
or	Логическое сложение