1. Встроенные типы данных Python и операции с ними

1.1 Знакомство с Python

Динамическая типизация как один из важнейших аспектов программирования на Python

Python - это язык программирования, который отличается простотой синтаксиса и гибкой системой работы с данными. Одной из ключевых черт Python является динамическая типизация. Это означает, что при создании переменной не нужно заранее указывать, какой тип данных она будет содержать. Тип определяется автоматически в момент присваивания значения.

```
a = 8
```

В этот момент создаётся объект типа "целое число", а имя a начинает ссылаться на этот объект. Если затем выполнить:

```
a = "текст"
```

Теперь a будет указывать уже на строковый объект. Таким образом, переменная в Python - это просто имя, которое может быть связано с объектом любого типа. В отличие от языков с жёсткой типизацией, здесь нет необходимости объявлять тип переменной заранее.

Когда мы связываем с переменной некоторое значение, просто переносим указатель на другой объект. Python предоставляет широкий набор встроенных типов данных, которые позволяют удобно работать с числами, текстом и коллекциями данных. Вот некоторые из них (табл.1.1):

Таблица 1.1 – Встроенные типы данных (часть):

Название типа	Описание
`int`	Это функция, возвращающая целое число в десятичной системе счисления. Пример: 2, 4, 8, -10, -2
`float`	Это функция, возвращающая число с плавающей запятой. Пример: 2.6, -5.2
`str`	Это функция, возвращающая строку (неизменяемую последовательность символов)
`bool`	Это функция, возвращающая булево значение (True или False) для объекта
`list`	Функция, возвращающая изменяемую упорядоченную коллекцию объектов произвольных типов. Пример: [2, 2.4, “Hello”]
`tuple`	Функция, возвращающая неизменяемую упорядоченную коллекцию объектов произвольных типов. Кортеж. Пример:(2, 2.4, “Hello”)

`dict`	Функция, возвращающая неупорядоченную коллекцию произвольных объектов с доступом по ключу. Пример: {“name”: “Вася”, “age”: 10}

Механизмы реализации ввода/вывода данных

Часто программа должна получать данные от пользователя и выводить результаты. Для ввода информации в Python используется функция **input**(). Она ожидает, что пользователь что-то напечатает, и возвращает введённую строку.

```
name = input("Как вас зовут? ")
```

После выполнения этой строки программа остановится и будет ждать, пока пользователь введёт своё имя. Всё, что введёт пользователь, сохранится в переменной **name** как строка.

Если нужно получить число, результат работы **input**() можно преобразовать с помощью функции int() или float():

```
age = int(input("Введите ваш возраст: "))
```

Для вывода информации на экран применяется функция **print**(). Она может принимать несколько аргументов и выводить их через пробел:

```
print("Меня зовут", name)
```

Арифметические и логические операции в Python

Python поддерживает стандартные арифметические операции (табл. 1.2):

Таблица 1.2 – Арифметические операторы в Python

Оператор	Описание	Примеры

+	Сложение	`x = 10`
		`y = 3`
		`print(x + y)  # 13`

-	Вычитание	`print(x - y)  #7`
*	Умножение	`print(x * y)  #30`
/	Деление	`print(x / y)  # 3.333`
//	Целочисленное деление	`print(x // y)  # 3`
%	Остаток от деления	`print(x % y)  # 1`
**	Возведение в степень	`print(x ** y)  # 1000`

Обратите внимание на операцию целочисленного деления с участием отрицательных чисел в качестве делимого или делителя. Сравним два примера:

Таблица 1.3 – Сравнение обычного деления и целочисленного

Пример	Результат
Деление	
`print(9 / 4)`	2.25
`print(-9 / 4)`	-2.25
Целочисленное деление	
`print(9 // 4)`	2
`print(-9 // 4)`	-3

Такой результат обусловлен тем, что целочисленное деление в Python 3 округляет итоговое значение в меньшую сторону. То есть для числа 2.25 это 2, а для числа -2.25 – -3.

Условные конструкции и циклы

Оператор if. Оператор **if** называют инструкцией. В качестве выражения может выступать любое выражение, которое будет автоматически преобразовано в логическое.

Синтаксис выглядит следующим образом:

```
if condition:
    #block of statements
else:
    #another block of statements (else-block)
```

Оператор if-else. Если выражение истинно (**True**), то выполняется «Блок if», при ложном выражении (**False**) – «Блок else». То есть выполняется либо первый блок, либо второй. Программа для проверки того, является ли число положительным или нет:

```
num = int(input("Введите число: "))
if num > 0:
    print("Число положительное")
else:
    print("Число не положительное")
```

Оператор elif. Оператор **elif** переводится как «иначе если». Логическое выражение, стоящее после оператора elif, проверяется, только если все вышестоящие условия ложные. То есть в этой схеме может выполниться только один блок кода: первый, второй, третий или четвёртый. Если одно из выражений истинно, то нижестоящие условия проверяться не будут.

Синтаксис:

```
if expression 1:
    # block of statements

elif expression 2:
    # block of statements

elif expression 3:
    # block of statements

else:
    # block of statements
```

Пример:

Чтобы проверялись все условия, независимо от результата предыдущего, следует использовать несколько независимых операторов **if**.

Рассмотрим ещё один пример:

```
if num > 0:
    print("Положительное")
elif num == 0:
    print("Ноль")
else:
    print("Отрицательное")
```

Оператор case. Оператор match-case, появившийся в Python 3.10, представляет собой современный инструмент для управления выполнением программы. Он позволяет лаконично и понятно обрабатывать различные варианты значений, напоминая конструкции switch-case из других языков, например, C или JavaScript, но с рядом особенностей, присущих Python.

В отличие от традиционных цепочек if-elif-else, match-case делает код более читаемым и структурированным, что особенно удобно при работе с большим количеством условий. Кроме того, этот оператор поддерживает работу с различными типами шаблонов: можно сопоставлять не только простые значения, но и списки, кортежи, словари или даже сложные структуры данных. Благодаря этому обработка вариантов становится более гибкой и менее подверженной ошибкам.

В конструкции match-case не требуется использовать break по окончании каждого блока case, что отличает её от switch-case в других языках. Также шаблоны могут быть не только литералами, но и более сложными структурами, что позволяет реализовывать мощные сценарии сопоставления данных

```
match выражение:
    case шаблон1:
        # действия, если шаблон1 совпадает
    case шаблон2:
        # действия, если шаблон2 совпадает
```

```
    case _:
        # действия по умолчанию, если ни один шаблон не совпал
```

Примеры использования оператора case

```
def describe_number(num):
    match num:
        case 1:
            return "Это единица"
        case 2:
            return "Это двойка"
        case _:
            return "Это какое-то другое число"

print(describe_number(1))  # Выведет: Это единица
print(describe_number(3))  # Выведет: Это какое-то другое число
```

Цикл задаёт многократное выполнение оператора.

Цикл while (“пока”) позволяет выполнить одну и ту же последовательность действий, пока проверяемое условие истинно. Условие записывается до тела цикла и проверяется до выполнения тела цикла. Как правило, цикл while используется, когда невозможно определить точное значение количества проходов исполнения цикла.

Синтаксис цикла while в простейшем случае выглядит так:

```
while условие:
    блок инструкций
```

При выполнении цикла `while` сначала проверяется условие. Если оно ложно, то выполнение цикла прекращается и управление передается на следующую инструкцию после тела цикла `while`. Если условие истинно, то выполняется инструкция, после чего условие проверяется снова и снова выполняется инструкция. Так продолжается до тех пор, пока условие будет истинно. Как только условие станет ложно, работа цикла завершится и управление передастся следующей инструкции после цикла.

Один шаг цикла (однократное выполнение тела цикла) называется **итерацией.** Пример цикла:

```
number = int(input('Введите целое число от 0 до 9: '))
while number < 10:
    print(number)
    number = number + 1
print('программа завершена успешно')
```

Цикл **for.** Цикл for позволяет удобно проходить по элементам любых итерируемых объектов: списков, строк, кортежей и других. Синтаксис:

```
for переменная in последовательность:
    # действия с переменной
```

Пример использования. Перебор элементов списка:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Результат:

```
apple
banana
cherry
```

Инструкции **break, continue.** В теле цикла можно использовать вспомогательные инструкции **break** и **continue**. Иногда применение этих инструкций позволяет упростить ваш код и сделать его более читабельным.

Оператор **continue** начинает следующий проход цикла, минуя оставшееся тело цикла.

Оператор **break** досрочно прерывает цикл. Пример:

```
i = 0
while True:
    i += 1
    if i >= 10:
    # инструкция break при выполнении немедленно заканчивает
выполнение цикла
      break
    if i % 2 == 0:
    # переходим к проверке условия цикла,
    # пропуская все операторы за инструкцией
      continue
    print(i)
    #i += 1
```

1.2 Встроенные типы и операции с ними

Тип данных: число

В Python есть несколько разновидностей числовых данных: целые числа (**int**), числа с плавающей точкой (**float**) и комплексные числа (**complex**).

Целые (int). Целые числа (int) позволяют выполнять стандартные арифметические действия, а также операции по модулю и побитовые вычисления. Например, чтобы получить модуль числа, используют функцию abs():

```
print(abs(-7))  # Выведет: 7
```

Для работы с битами доступны операции "И" (&), "ИЛИ" (|), "исключающее ИЛИ" (^), а также сдвиги влево (<<) и вправо (>>):

```
print(5 & 3)    # 1
print(5 | 3)    # 7
print(5 ^ 3)    # 6
print(5 << 2)   # 20
print(5 >> 1)   # 2
```

Python позволяет представлять числа не только в десятичной системе, но и в двоичной, восьмеричной и шестнадцатеричной. Для преобразования между системами счисления используются функции bin(), oct(), hex():

```
print(bin(10))   # 0b1010
print(oct(10))   # 0o12
print(hex(10))   # 0xa
```

Вещественные (float). Поддерживают операции, аналогичные операциям, которые выполняются с целыми числами. Более подробно рассмотрены в ранее.

Комплексные (complex). Под комплексным числом понимается выражение вида **a + ib**, где **a** и **b** – любые действительные числа, **i** – мнимая единица.

```
z = complex(3, 4)
print(z)  # (3+4j)
```

Тип данных: строка

Строка в Python — упорядоченный набор символов для хранения и представления текстовой информации. Пример:

```
text = "Пример строки"
print(text)
```

Простейший тип данных – упорядоченная неизменяемая коллекция элементов. Со строками в Python можно выполнять множество операций, например:

Конкатенация (сцепление). Пример:

```
s1 = 'hello '
s2 = 'world'
print(s1 + s2)
```

Результат:

```
hello world
```

Взятие элемента по индексу. Пример:

```
s = hello world
print(s[1])
```

Результат:

```
e
```

Извлечение среза. Синтаксис: **[s:f:step]**, где **s** – начало среза, **f** – окончание, **step** – шаг (опционально).Пример:

```
s = 'abrakadabra'
print("1-", s[4:7])
print("2-", s[3:-3])
print("3-",s[:5])
print("4-",s[3:])
print("5-",s[:])
print("6-",s[::-1])
print("7-",s[1:7:2])
```

Результат:

```
1- kad
2- akada
3- abrak
4- akadabra
5- abrakadabra
6- arbadakarba
7- baa
```

Рассмотрим методы, применяемые в приложениях для операций со строками, и примеры их использования (табл.1.4).

Таблица 1.4 – Методы строк

Функция	Описание	Пример
`len(строка)`	Возвращает длину строки	`print(len("my_string"))` `-> 9`

`строка.split(<разделитель>)`	Разбить строку по разделителю	`print("раз два три".split())`
		`-> ['раз', 'два', 'три']`
		`print("четыре_пять_шесть".split('_'))`
		`-> ['четыре', 'пять', 'шесть']`

<разделитель>.join (список)	Собрать строку из списка с указанным разделителем	print('_'.join(['раз', 'два', 'три'])) -> раз_два_три print(''.join(['раз', 'два', 'три'])) -> раздватри
строка.title()	Перевести первую букву каждого слова в верхний регистр, остальные - в нижний	print("лето быстро пролетело".title()) -> Лето Быстро Пролетело
строка.upper()	Преобразовать строку к верхнему регистру	print('простая строка'.upper()) -> ПРОСТАЯ СТРОКА
строка.lower()	Преобразовать строку к нижнему регистру	print('ПРОСТАЯ СТРОКА'.lower()) -> простая строка
строка.istitle()	Проверить, начинаются ли слова строки с буквы в верхнем регистре	print('Лето Быстро Пролетело'.istitle()) -> True print('Лето быстро Пролетело'.istitle()) -> False
строка.isupper()	Проверить, состоит ли строка из символов в верхнем регистре	print('ПРОСТАЯ СТРОКА'.isupper()) -> True print('простая строка'.isupper()) -> False
строка.islower()	Проверить, состоит ли строка из символов в нижнем регистре	print('простая строка'.islower()) -> True print('ПРОСТАЯ СТРОКА'.islower()) -> False
ord(символ)	Получить ASCII-код для символа	print(ord(',')) -> 44
chr(код)	Получить символ по ASCII-коду	print(chr(44)) -> ','
строка.count(подстрока, [начало], [конец])	Вернуть количество вхождений	print('lalala'.count('la')) -> 3 print('lalala'.count('la', 2, 4)) -> 1

	подстроки в строку	
`строка.capitalize()`	Перевести первый символ строки в верхний регистр, остальные - в нижний	`print('cTPOKA'.capitalize()) -> Строка`
`строка.startswith(шаблон)`	Проверить, начинается ли строка с шаблона	`print('papapa'.startswith('pa')) -> True` `print('papapa'.startswith('не')) -> False`
`строка.endswith(шаблон)`	Проверить, заканчивается ли строка шаблоном	`print('papapa'.endswith('pa')) -> True` `print('papapa'.endswith('не')) -> False`
`строка.replace(шаблон, замена)`	Заменить в строке шаблон на указанную подстроку	`print('lalala'.replace('la', 'na')) -> 'nanana'`

`строка.index(подстрока, [начало], [конец])`	Найти подстроку в строке. Получить позицию первого вхождения или получить ValueError	`print('lalalala'.index('la')) -> 0`
		`print('lalalala'.index('la', 4, 6)) -> 4`
		`print('lalalala'.index('la', 10, 20)) -> ValueError: substring not found`

`строка.find(подстрока, [начало], [конец])`	Найти подстроку в строке. Получить позицию первого вхождения или получить -1	`print('lalalala'.find('la')) -> 0`
		`print('lalalala'.find('la', 4, 6)) -> 4`
		`print('lalalala'.find('la', 10, 20)) -> -1`

Тип данных: список

В Python массивов как таковых нет. Список (list) - это изменяемая последовательность элементов, которые могут быть любого типа. Списки создаются с помощью квадратных скобок:

```
numbers = [1, 2, 3, 4]
fruits = ["яблоко", "банан", "груша"]
```

Списки поддерживают индексацию, срезы, добавление и удаление элементов:

```
numbers.append(5)      # Добавить элемент в конец
numbers.remove(2)      # Удалить первое вхождение элемента
print(numbers[0])      # Получить первый элемент
print(numbers[1:3])    # Получить срез списка
```

result_list.extend(). Дополняет список элементами из указанного объекта. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# extend
result_list.extend([8, 9, 10])
print(result_list)
```

Результат:

```
[2, 'text', 456, 45.3, None, 8, 9, 10]
```

result_list.insert(i, x). Вставляет указанный элемент перед указанным индексом. **i** – позиция (индекс), перед которой требуется поместить элемент. Нумерация ведётся с нуля. Поддерживается отрицательная индексация. **x** – Элемент, который требуется поместить в список. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# insert
result_list.insert(1, "ins_el")
print(result_list)
```

Результат:

```
[2, 'ins_el', 'text', 456, 45.3, None]
```

result_list.remove(x).Удаляет из списка указанный элемент. x – элемент, который требуется удалить из списка. Если элемент отсутствует в списке, возбуждается ValueError. Удаляется только первый обнаруженный в списке элемент, значение которого совпадает со значением переданного в метод. Пример:

```
result_list = [2, 'text', 456, 45.3, None, "ins_el"]

# remove
result_list.remove("ins_el")
```

```
print(result_list)
```

Результат:

```
[2, 'text', 456, 45.3, None]
```

result_list.pop(i).Возвращает элемент [на указанной позиции], удаляя его из списка. **i=None** – Позиция искомого элемента в списке (целое число). Если не указана, считается что имеется в виду последний элемент списка. Отрицательные числа поддерживаются. Чтобы удалить элемент из списка не возвращая его, воспользуйтесь list.remove(). Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# pop
result_list.pop(1)
print(result_list)
```

Результат:

```
[2, 456, 45.3, None]
```

result_list.index(x[, start[, end]]). Метод возвращает положение первого индекса, со значением **x**. Также можно указать границы поиска **start** и **end**. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# index
print(result_list.index(None))
```

Результат:

```
4
```

result_list.count(x). Метод возвращает положение первого индекса, со значением **x**. Также можно указать границы поиска **start** и **end**. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# count
print(result_list.count(2))
```

Результат:

```
1
```

result_list.reverse(). Перестраивает элементы списка в обратном порядке. Внимание: Данный метод модифицирует исходный объект на месте, возвращая при этом **None**. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# reverse
result_list.reverse()
print(result_list)
```

Результат:

```
[None, 45.3, 456, 'text', 2]
```

result_list.copy(). Возвращает копию списка. Внимание: Возвращаемая копия является поверхностной (без рекурсивного копирования вложенных элементов). Действие метода эквивалентно выражению my_list[:]. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# copy
copy_list = result_list.copy()
print(copy_list)
```

Результат:

```
[2, 'text', 456, 45.3, None]
```

result_list.clear(). Удаляет из списка все имеющиеся в нём значения. Действие метода эквивалентно выражению del my_list[:]. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# clear
result_list.clear()
print(result_list)
```

Результат:

```
[]
```

Тип данных: кортеж

Кортеж (tuple) - это неизменяемая последовательность элементов. Кортежи создаются с помощью круглых скобок:

```
point = (2, 3)
colors = ("красный", "зеленый", "синий")
```

Кортежи удобны для хранения фиксированных наборов данных, которые не должны изменяться. Как коллекции, поддерживают те же

операции, что и списки. Операции не должны изменять саму коллекцию (например, **index()**, **count()**).

Тип данных: множество

Множество (set) - это коллекция уникальных элементов без определенного порядка. Множества создаются с помощью фигурных скобок или функции set():

Пример:

```
unique_numbers = {1, 2, 3, 3, 2}
print(unique_numbers)  # {1, 2, 3}
```

Множества поддерживают операции объединения, пересечения и разности:

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a | b)  # {1, 2, 3, 4, 5}
print(a & b)  # {3}
print(a - b)  # {1, 2}
```

Далее будут рассмотрены методы работы с изменяемыми множествами. Зададим множество. Пример:

```
my_set = {400, None, "text", True}
print(my_set)
```

Результат:

```
{400, True, None, 'text'}
```

set.add(x). Добавляет элемент в множество. Пример:

```
my_set = {400, None, "text", True}

# add
my_set.add("another_el")
print(my_set)
```

Результат:

```
{True, 'another_el', 'text', 400, None}
```

set.remove(x). Удаляет элемент из множества. KeyError, если такого элемента не существует. Пример:

```
my_set = {400, None, "text", True}

# remove
my_set.remove("text")
```

```
print(my_set)
```

Результат:

```
{400, True, None}
```

set.discard(x). Удаляет элемент, если он находится в множестве. Пример:

```
my_set = {400, None, "text", True}

# discard
my_set.discard(400)
print(my_set)
```

Результат:

```
{True, 'text', None}
```

set.pop(). Удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым. Пример:

```
my_set = {400, None, "text", True}

# pop
my_set.pop()
print(my_set)
```

Результат:

```
{True, 'text', None}
```

set.copy(). Копия множества. Пример:

```
my_set = {400, None, "text", True}

# copy
print(my_set.copy())
```

Результат:

```
{400, 'text', None, True}
```

set.clear(). Очистка множества. Пример:

```
my_set = {400, None, "text", True}

# clear
my_set.clear()
print(my_set)
```

Результат:

```
set()
```

Изменяемые множества (**set()**) и неизменяемые (**frozenset()**) – аналогия списков и кортежей. Пример:

```
my_s = set('abrakadabra')
my_fs = frozenset('abrakadabra')
print(my_s == my_fs)
```

Результат:

```
True
```

Пример:

```
# вычитание
my_s = set('kadabra')
print(my_s)
my_fs = frozenset('abra')
print(my_fs)
print(my_s - my_fs)
```

Результат:

```
{'r', 'a', 'b', 'd', 'k'}
frozenset({'r', 'a', 'b'})
{'k', 'd'}
```

Пример:

```
# объединение
my_s = set('kadabra')
print(my_s)
my_fs = frozenset('abra')
print(my_fs)
print(my_s | my_fs)
```

Результат:

```
{'b', 'r', 'a', 'd', 'k'}
frozenset({'b', 'a', 'r'})
{'b', 'd', 'r', 'k', 'a'}
```

Тип данных: словарь

Словарь (dict) - это коллекция пар "ключ-значение". Ключи должны быть уникальными, а значения могут быть любыми:

```
person = {"имя": "Анна", "возраст": 25}
```

```
print(person["имя"])  # Анна
```

Словари позволяют быстро находить значения по ключу и изменять их:

```
person["город"] = "Москва"  # Добавить новый ключ
del person["возраст"]       # Удалить ключ
```

dict.keys(). Возвращает ключи в словаре. Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# keys
print(my_dict.keys())
```

Результат:

```
dict_keys(['key_1', 2, 'key_3', 4])
```

dict.values(). Возвращает значения в словаре. Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# values
print(my_dict.values())
```

Результат:

```
dict_values([500, 400, True, None])
```

dict.items(). Возвращает пары (ключ, значение). Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# items
print(my_dict.items())
```

Результат:

```
dict_items([('key_1', 500), (2, 400), ('key_3', True), (4,
None)])
```

dict.get(key[, default]). Возвращает значение ключа, но если его нет, не бросает исключение, а возвращает **default** (по умолчанию **None**). Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# get
print(my_dict.get(2))
```

Результат:

```
400
```

dict.popitem(). Удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение **KeyError**. Помните, что словари неупорядочены. Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# popitem
print(my_dict.popitem())
print(my_dict.popitem())
print(my_dict.popitem())
print(my_dict.popitem())
```

Результат:

```
(4, None)
('key_3', True)
(2, 400)
('key_1', 500)
```

dict.setdefault(key[, default]). Возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ со значением **default** (по умолчанию **None**). Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# setdefault
print(my_dict.setdefault(5))
print(my_dict.items())
```

Результат:

```
None
dict_items([('key_1', 500), (2, 400), ('key_3', True), (4, None),
(5, None)])
```

dict.pop(key[, default]). Удаляет ключ и возвращает значение. Если ключа нет, возвращает **default** (по умолчанию бросает исключение). Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# pop
print(my_dict.pop(2))
print(my_dict.items())
```

Результат:

```
400
dict_items([('key_1', 500), ('key_3', True), (4, None)])
```

dict.update([other]). Обновляет словарь, добавляя пары (ключ, значение) из **other**. Существующие ключи перезаписываются. Возвращает **None** (не новый словарь!). Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# update
my_dict.update({8: 8, 9: 9, 10: 10})
print(my_dict.items())
```

Результат:

```
dict_items([('key_1', 500), (2, 400), ('key_3', True), (4, None),
(8, 8), (9, 9), (10, 10)])
```

dict.copy(). Возвращает копию словаря. Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}
# copy
print(my_dict.copy())
```

Результат:

```
{'key_1': 500, 2: 400, 'key_3': True, 4: None}
```

dict.clear(). Очищает словарь. Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# clear
my_dict.clear()
print(my_dict.items())
```

Результат:

```
dict_items([])
```

Тип данных: bool

Тип bool используется для хранения значений True и False. Логические значения часто используются в условиях и циклах:

```
is_active = True
if is_active:
    print("Активно")
```

Функция **bool()** позволяет привести любое значение к логическому типу, если это значение может быть интерпретировано в качестве логического типа.

Байтовые последовательности

Типы bytes и bytearray предназначены для работы с бинарными данными:

```
data = bytes([50, 100, 76])
print(data)  # b'2dL'
```

Тип данных: NoneType

Тип NoneType представлен единственным объектом None, который используется для обозначения отсутствия значения:

```
result = None
if result is None:
    print("Значение не определено")
```

Рассмотрим ещё один пример:

```
my_dict = {'name': 'Ivan', 'surname': 'Ivanov', 'age': 40,
'position': None}
for el in my_dict:
    if my_dict[el] == None:
        print(f"Для сотрудника пока не определён параметр: {el}")
```

Результат:

```
Для сотрудника пока не определён параметр: position
```

Здесь выполняется перебор ключей словаря и проверка, есть ли в словаре значения типа None.

Оператор is

Проверяет тождественность (идентичность) двух объектов в памяти. Возвращает значение **True** (истина), если переменные ссылаются на один и тот же объект.

Пример:

```
a = 20
b = 20

if a is b:
    print("Переменные идентичны")
else:
    print("Переменные не идентичны")
```

Результат:

```
Переменные идентичны
```

Важная особенность использования оператора **is** заключается в том, что он не идентичен оператору ==.

== – проверка равенства значений двух объектов.

is – проверка идентичности объектов, то есть проверка того, что переменные указывают на один и тот же объект в памяти. Пример:

```
obj_1 = [10, 20, 30, 40]
obj_2 = obj_1
print(obj_1 == obj_2)
print(obj_1 is obj_2)

obj_2 = obj_1[:] # переменная obj_2 ссылается на копию obj_1
print(obj_1 == obj_2)
print(obj_1 is obj_2)
print(obj_1 is not obj_2)
```

Результат:

```
True
True
True
False
True
```

Для проверки соответствия объекта типу **NoneType** предпочтительно использовать оператор **is**. Пример:

```
obj_1 = None
print(obj_1 is None)
```

Результат:

```
True
```

1.3 Полезные советы для работы в Python

В завершение урока познакомимся с набором интересных приёмов, которые пригодятся вам на практике.

Объединение списков без цикла. Явный вариант решения задачи объединения списков разной длины предполагает перебор элементов в цикле. Но возможно и более лаконичное решение через функцию **sum()**. Пример:

```
my_list = [[10, 20, 30], [40, 50], [60], [70, 80, 90]]
print(sum(my_list, []))
```

Результат:

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

Поиск уникальных элементов в списке. Это очень популярный трюк, предполагающий трансформацию списка во множество и поиск уникальных элементов. Пример:

```
my_list = [10, 10, 3, 4, 5, 9, 30, 30]
```

```
print(list(set(my_list)))
```

Результат:

```
[3, 4, 5, 9, 10, 30]
```

Обмен значениями через кортежи. Позволяет выполнять обмен значения без создания дополнительной переменной. Трюк допустим для любого числа переменных. Пример:

```
var_1, var_2 = 20, 30
print(var_1, var_2)
var_1, var_2 = var_2, var_1
print(var_1, var_2)
```

Результат:

```
20 30
30 20
```

Правая часть выражения может представлять собой любой итерируемый объект. Главное, чтобы число элементов в левой и правой частях совпадало.

Вывод значения несуществующего ключа в словаре. Если попытаться обратиться к несуществующему ключу словаря, возникнет исключение. Пример:

```
my_dict = {'k_1': 20, 'k_2': True, 'k_3': 'text'}
print(my_dict['k_4'])
```

Результат:

```
KeyError: 'k_4'
```

Чтобы избежать такую ситуацию, можно воспользоваться методом **get()**. Пример:

```
my_dict = {'k_1': 20, 'k_2': True, 'k_3': 'text'}
print(my_dict.get('k_4'))
```

Результат:

```
None
```

Поиск самых часто встречающихся элементов списка. Искать самый часто встречающийся элемент можно, используя встроенную функцию **max()**. Она ищет наибольшее значение не только для итерируемого объекта, но и для результатов применения к этому объекту функции. Можно преобразовать

список во множество и применить метод **count** для определения количества вхождений элемента в итерируемый объект. Пример:

```
my_list = [10, 20, 20, 20, 30, 50, 70, 30]
print(max(set(my_list), key=my_list.count))
```

Результат:

```
20
```

Вывод с помощью функции print() без перевода строки. По умолчанию функция **print()** добавляет символ перевода строки, который можно отменить, добавив в функцию параметр **end** со значением пустой строки. Пример:

```
for el in ["ab", "ra", "kada", "bra"]:
    print(el, end='')
```

Результат:

```
abrakadabra
```

Сортировка словаря по значениям. По умолчанию элементы словаря сортируются по наименованиям ключей. Пример:

```
my_dict = {'python': 1991, 'java': 1995, 'c++': 1983}
print(sorted(my_dict))
```

Результат:

```
['c++', 'java', 'python']
```

Но можно реализовать сортировку по значениям элементов. Пример:

```
my_dict = {'python': 1991, 'java': 1995, 'c++': 1983}
print(sorted(my_dict, key=my_dict.get))
```

Результат:

```
['c++', 'python', 'java']
```

Нумерованные списки. Для реализации нумерованного списка можно воспользоваться функцией **enumerate()**. Пример:

```
for ind, el in enumerate(['ноль', 'один', 'два', 'три']):
    print(ind, el)
```

Результат:

```
0 ноль
```

```
1 один
2 два
3 три
```

Пример:

```
for ind, el in enumerate(['один', 'два', 'три'], 1):
    print(ind, el)
```

Результат:

```
1 один
2 два
3 три
```