Лабораторная работа №1

Тема: «Работа со встроенными типами данных, числовые типы, строки, кортежи, измененяемые последовательности. Применение основных арифметических операций, определение приоритетов».

Арифметические операции

Python поддерживает все распространенные арифметические операции.

Сложение:

```
print(6 + 2)  # 8
```

Вычитание:

```
print(6 - 2)  # 4
```

Умножение:

```
print(6 * 2)  # 12
```

Деление:

```
print(6 / 2)  # 3.0
```

Целочисленное деление:

```
print(7 / 2)  # 3.5
print(7 // 2)  # 3 (целочисленный результат деления)
```

Возведение в степень:

```
print(6 ** 2)  # Возводим число 6 в степень 2. Результат - 36
```

Получение остатка от деления:

```
print(7 % 2)  # 1
```

При последовательном использовании нескольких арифметических операций их выполнение производится в соответствии с их приоритетом. В начале выполняются операции с большим приоритетом. Приоритеты операций в порядке убывания приведены в следующей таблице.

Таблица 1. Приоритеты операций

Операции	**Направление**
**	Справа налево
* / // %	Слева направо
+ -	Слева направо

```
number = 3 + 4 * 5 ** 2 + 7
print(number)  # 110
```

```
number = (3 + 4) * (5 ** 2 + 7)
print(number)  # 224
```

Следует отметить, что в арифметических операциях могут принимать участие как целые, так и дробные числа. Если в одной операции участвует целое число (***int***) и число с плавающей точкой (***float***), то целое число приводится к типу ***float***.

Арифметические операции с присвоением

```
number = 10
number += 5
print(number)  # 15

number -= 3
print(number)  # 12

number *= 4
print(number)  # 48
```

Функции преобразования чисел

```
a = "2"
b = 3
c = int(a) + b
print(c) # 5
```

Аналогичным образом действует функция ***float()***, которая преобразует аргумент в число с плавающей точкой.

Представление числа

При обычном определении числовой переменной она получает значение в десятичной системе. Но кроме десятичной в Python мы можем использовать двоичную, восьмеричную и шестнадцатеричную системы.

Для определения числа в двоичной системе перед его значением ставится 0 и префикс b:

```
x = 0b101   # 101 в двоичной системе равно 5
```

Для определения числа в восьмеричной системе перед его значением ставится 0 и префикс o:

```
a = 0o11    # 11 в восьмеричной системе равно 9
```

Для определения числа в шестнадцатеричной системе перед его значением ставится 0 и префикс x:

```
y = 0x0a    # a в шестнадцатеричной системе равно 10
```

И с числами в других системах измерения также можно проводить арифметические операции:

```
x = 0b101   # 5
y = 0x0a    # 10
z = x + y   # 15
print("{0} in binary {0:08b} in hex {0:02x} in octal {0:02o}".format(z))
```

Для вывода числа в различных системах исчисления используются функция ***format***, которая вызывается у строки. В эту строку передаются различные форматы. Для двоичной системы **"{0:08b}"**, где число **8** указывает, сколько знаков должно быть в записи числа. Если знаков указано больше, чем требуется для числа, то ненужные позиции заполняются нулями. Для шестнадцатеричной системы применяется формат **"{0:02x}"**. И здесь все аналогично – запись числа состоит из двух знаков, если один знак не нужен, то вместо него вставляется ноль. А для записи в восьмеричной системе используется формат **"{0:02o}"**.

Результат работы скрипта:

```
15 in binary 00001111   in hex 0f   in octal 17
```

Условные выражения

Ряд операций представляют условные выражения. Все эти операции принимают два операнда и возвращают логическое значение, которое в ***Python*** представляет тип ***boolean***. Существует только два логических значения – ***True*** (выражение истинно) и ***False*** (выражение ложно).

Операции сравнения

Простейшие условные выражения представляют операции сравнения, которые сравнивают два значения. Python поддерживает следующие операции сравнения:

Таблица 2. Операции сравнения

Оператор	Операция сравнения
==	Возвращает True, если оба операнда равны. Иначе возвращает False.
!=	Возвращает True, если оба операнда НЕ равны. Иначе возвращает False.
>	Возвращает True, если первый операнд больше второго.
<	Возвращает True, если первый операнд меньше второго.
>=	Возвращает True, если первый операнд больше или равен второму.
<=	Возвращает True, если первый операнд меньше или равен второму.

Примеры операций сравнения:

```
a = 5
b = 6
result = 5 == 6  # сохраняем результат операции в переменную
print(result)  # False - 5 не равно 6
print(a != b)  # True
print(a > b)  # False - 5 меньше 6
print(a < b)  # True

bool1 = True
bool2 = False
print(bool1 == bool2)  # False - bool1 не равно bool2
```

Операции сравнения могут сравнивать различные объекты – строки, числа, логические значения, однако оба операнда операции должны представлять один и тот же тип.

Логические операции

Для создания составных условных выражений применяются логические операции. В Python имеются следующие логические операторы:

and (логическое умножение)

Возвращает ***True***, если оба выражения равны ***True***

```
age = 22
weight = 58
result = age > 21 and weight == 58
print(result)  # True
```

В данном случае оператор ***and*** сравнивает результаты двух выражений: `age > 21` `weight == 58`. И если оба этих выражений возвращают ***True***, то оператор ***and*** также возвращает ***True***. Причем в качестве одного из выражений необязательно выступает операция сравнения: это может быть другая логическая операция или просто переменная типа ***boolean***, которая хранит ***True*** или ***False***.

```
age = 22
weight = 58
isMarried = False
result = age > 21 and weight == 58 and isMarried
print(result)  # False, так как isMarried = False
```

or (логическое сложение)

Возвращает ***True***, если хотя бы одно из выражений равно ***True***

```
age = 22
isMarried = False
result = age > 21 or isMarried
print(result)  # True, так как выражение age > 21 равно True
```

not (логическое отрицание)

Возвращает ***True***, если выражение равно ***False***

```
age = 22
isMarried = False
print(not age > 21)  # False
print(not isMarried)  # True
```

Если один из операндов оператора ***and*** возвращает ***False***, то другой операнд уже не оценивается, так как оператор в любом случае возвратит ***False***. Подобное поведение

позволяет немного увеличить производительность, так как не приходится тратить ресурсы на оценку второго операнда.

Аналогично если один из операндов оператора ***or*** возвращает ***True***, то второй операнд не оценивается, так как оператор в любом случае возвратит ***True***.

Если в одном выражении одновременно используется несколько или даже все логические операторы, то следует учитывать, что они имеют разные приоритеты. Вначале выполняется оператор ***not***, затем оператор ***and***, а в конце оператор ***or***:

```
age = 22
isMarried = False
weight = 58
result = weight == 58 or isMarried and not age > 21  # True
print(result)
```

Здесь будет следующий ход вычислений:

1. not age > 21 равно False
2. isMarried and False (not age > 21) равно False
3. weight == 58 or False (isMarried and not age > 21) равно True

Для переопределения порядка вычислений мы можем использовать скобки:

```
age = 22
isMarried = False
weight = 58
result = (weight == 58 or isMarried) and not age > 21  # False
print(result)
```

В отличие от первого случая здесь результатом будет значение ***False***.

Операции со строками

Строка представляет последовательность символов в кодировке ***Unicode***, заключенных в кавычки. Строковые выражения могут заключаться в одинарные, двойные и тройные кавычки.

```
name = "Tom"
surname = 'Smith'
print(name, surname)  # Tom Smith

print("""
Многострочное
Текстовое
Сообщение
""")
```

Также может встречаться комбинация из одинарных и двойных кавычек. Строковое выражение, обрамленное в тройные кавычки, может состоять из нескольких строчек. В случае, когда используются комбинации из двойных и одинарных кавычек, одна пара должна быть внешней – она обозначает начало и конец строкового выражения, а внутренняя пара является частью этого выражения.

Одной из самых распространенных операций со строками является их объединение или конкатенация. Для объединения строк применяется знак плюса:

```
name = "Tom"
surname = 'Smith'
fullname = name + " " + surname
print(fullname)  # Tom Smith
```

С объединением двух строк все просто, но что, если нам надо сложить строку и число? В этом случае необходимо привести число к строке с помощью функции ***str()***:

```
name = "Tom"
age = 33
info = "Name: " + name + " Age: " + str(age)
print(info)  # Name: Tom Age: 33
```

Escape-последовательности (*Escape Sequence*)

Кроме стандартных символов строки могут включать управляющие *escape*-последовательности, которые интерпретируются особым образом. Например, последовательность `\n` представляет перевод строки. Поэтому следующее выражение:

```
print("Время пришло в гости отправиться\nждет меня старинный друг")
```

На консоль выведет две строки:

```
Время пришло в гости отправиться
ждет меня старинный друг
```

То же самое касается и последовательности `\t`, которая добавляет табуляцию.

Кроме того, существуют символы, которые вроде бы сложно использовать в строке. Например, кавычки. И чтобы отобразить кавычки (как двойные, так и одинарные) внутри строки, перед ними ставится слеш:

```
print("Кафе \"Central Perk\"")
```

Подробнее: https://docs.python.org/3.11/reference/lexical_analysis.html

Сравнение строк

Особо следует сказать о сравнении строк. При сравнении строк принимаются во внимание символы и их регистр. Так, цифровой символ условно меньше, чем любой алфавитный символ. Алфавитный символ в верхнем регистре условно меньше, чем алфавитные символы в нижнем регистре.

Например:

```
str1 = "1a"
str2 = "aa"
str3 = "Aa"
print(str1 > str2)  # False, так как первый символ в str1 - цифра
print(str2 > str3)  # True, так как первый символ в str2 - в нижнем регистре
```

Поэтому строка `"1a"` условно «меньше», чем строка `"aa"`. Вначале сравнение идет по первому символу. Если начальные символы обоих строк представляют цифры, то меньшей считается меньшая цифра, например, `"1a"` меньше, чем `"2a"`.

Если начальные символы представляют алфавитные символы в одном и том же регистре, то смотрят по алфавиту. Так, `"aa"` меньше, чем `"ba"`, а `"ba"` меньше, чем `"ca"`.

Если первые символы одинаковые, в расчет берутся вторые символы при их наличии.

Зависимость от регистра не всегда желательна, так как, по сути, мы имеем дело с одинаковыми строками. В этом случае перед сравнением мы можем привести обе строки к одному из регистров. Функция ***lower()*** приводит строку к нижнему регистру, а функция ***upper()*** – к верхнему.

```
str1 = "Tom"
str2 = "tom"
print(str1 == str2)  # False - строки не равны

print(str1.lower() == str2.lower())  # True
```

Списки, кортежи, словари и множества

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

Список (***list***) представляет тип данных, который хранит набор или последовательность элементов. Для создания списка в квадратных скобках ([]) через запятую перечисляются все его элементы. Во многих языках программирования есть аналогичная структура данных, которая называется массив.

Кортеж (***tuple***) представляет последовательность элементов, которая во многом похожа на список за тем исключением, что кортеж является неизменяемым (***immutable***) типом. Поэтому мы не можем добавлять или удалять элементы в кортеже, изменять его. Для создания кортежа используются круглые скобки, внутри которых через запятую перечисляются его элементы.

Пустой кортеж работает как синглтон, то есть в памяти запущенного Python скрипта всегда находится только один пустой кортеж. Все пустые кортежи просто ссылаются на один и тот же объект, это возможно благодаря тому, что кортежи неизменяемы. Такой подход сохраняет много памяти и ускоряет процесс работы с пустыми кортежами.

Чтобы избежать накладных расходов на постоянное изменение размера списков, Python не изменяет его размер каждый раз, как только это требуется. Вместо этого в каждом списке есть набор дополнительных ячеек, которые скрыты для пользователя, но в дальнейшем могут быть использованы для новых элементов. Как только скрытые ячейки заканчиваются, Python добавляет дополнительное место под новые элементы. Причём делает это с хорошим запасом, количество скрытых ячеек выбирается на основе текущего размера списка – чем он больше, тем больше дополнительных скрытых слотов под новые элементы.

Наряду со списками и кортежами ***Python*** имеет еще одну встроенную структуру данных, которая называется словарь (***dictionary***). В ряде языков программирования есть похожие структуры (словарь в C#, ассоциативный массив в PHP). Как и список, словарь хранит коллекцию элементов. Каждый элемент в словаре имеет уникальный ключ, с которым ассоциировано некоторое значение. Для создания словаря используют фигурные скобки, в которых через запятую указываются пары ключ-значение, разделенные двоеточием.

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

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

Подробнее о структурах данных –
https://docs.python.org/3.11/tutorial/datastructures.html,
Примеры – https://kshmirko.github.io/python/data-structures.html

Условная конструкция if

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

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

В самом простом виде после ключевого слова ***if*** идет логическое выражение. И если это логическое выражение возвращает ***True***, то выполняется последующий блок инструкций, каждая из которых должна начинаться с новой стоки и должна иметь отступы от начала строки:

```
age = 22
if age > 21:
    print("Доступ разрешен")
print("Завершение работы")
```

Поскольку в данном случае значение переменной ***age*** больше 21, то будет выполняться блок ***if***, а консоль выведет следующие строки:

```
Доступ разрешен
Завершение работы
```

Отступ желательно делать в 4 пробела (*устанавливается в настройках IDE*).

Обратите внимание в коде на последнюю стоку, которая выводит сообщение `"Завершение работы"`. Она не имеет отступов от начала строки, поэтому она не принадлежит к блоку ***if*** и будет выполняться в любом случае, даже если выражение в конструкции ***if*** возвратит ***False***.

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

```
age = 22
if age > 21:
    print("Доступ разрешен")
    print("Завершение работы")
```

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

```
age = 18
if age > 21:
    print("Доступ разрешен")
else:
    print("Доступ запрещен")
```

Если выражение `age > 21` возвращает ***True***, то выполняется блок ***if***, иначе выполняется блок ***else***. Если необходимо ввести несколько альтернативных условий, то можно использовать дополнительные блоки ***elif***, после которого идет блок инструкций.

```
age = 18
if age >= 21:
    print("Доступ разрешен")
elif age >= 18:
    print("Доступ частично разрешен")
else:
    print("Доступ запрещен")
```

Вложенные конструкции if

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

```
age = 18
if age >= 18:
    print("Больше 17")
    if age > 21:
        print("Больше 21")
    else:
        print("От 18 до 21")
```

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

```
age = 18
if age >= 18:
    print("Больше 17")
if age > 21:
    print("Больше 21")
else:
    print("От 18 до 21")
```

Теперь напишем небольшую программу, которая использует условные конструкции. Данная программа будет представлять собой своего рода обменный пункт:

```
# Программа Обменный пункт

usd = 65
euro = 73

money = int(input("Введите сумму, которую вы хотите обменять: "))
currency = int(input("Укажите код валюты (доллары - 400, евро - 401): "))

if currency == 400:
    cash = round(money / usd, 2)
    print("Валюта: доллары США")
elif currency == 401:
    cash = round(money / euro, 2)
    print("Валюта: евро")
else:
    cash = 0
    print("Неизвестная валюта")

print("К получению:", cash)
```

С помощью функции `input()` получаем вводимые пользователем данные на консоль. Причем данная функция возвращает данные в виде строки, поэтому нам надо ее еще привести к целому числу с помощью функции `int()`, чтобы введенные данные можно было использовать в арифметических операциях.

Программа подразумевает, что пользователь вводит количество средств, которые надо обменять, и код валюты, на которую надо произвести обмен. Коды валюты достаточно условны: 400 для долларов и 401 для евро.

С помощью конструкции ***if*** проверяем код валюты и делим на соответствующий валютный курс. Так как в процессе деления образуется довольно длинное число с плавающей точкой, которое может содержать множество знаков после запятой, то оно округляется до двух чисел после запятой с помощью функции `round()`.

В завершение на консоль выводится полученное значение. Например, запустим программу и введем данные:

```
Введите сумму, которую вы хотите обменять: 20000
Укажите код валюты (доллары - 400, евро - 401): 401
Валюта: евро
К получению: 273.97
```

Конструкция match/case

Оператор `match` принимает выражение и сравнивает его значение с шаблонами (case pattern). Синтаксис конструкции match/case:

```
match subject:
    case <pattern_1>:
        <action_1>
    case <pattern_2>:
        <action_2>
    case _:
        <action_wildcard>
```

Пример:

```
match status:
    case 400:
        print("Bad request")
    case 401 | 403 | 404:
        print("Not allowed")
    case 404:
        print("Not found")
    case 418:
        print("I'm a teapot")
    case _:
        print("Something's wrong with the internet")
```

Циклы

Циклы позволяют повторять некоторое действие в зависимости от соблюдения некоторого условия.

Цикл *while*

Первый цикл, который мы рассмотрим, это цикл ***while***. Он имеет следующее формальное определение:

```
while условное_выражение:
   инструкции
```

После ключевого слова ***while*** указывается условное выражение, и пока это выражение возвращает значение ***True***, будет выполняться блок инструкций, который идет далее.

Все инструкции, которые относятся к циклу ***while***, располагаются на последующих строках и должны иметь отступ от начала строки.

```
choice = "y"

while choice.lower() == "y":
    print("Привет")
    choice = input("Для продолжения нажмите Y, а для выхода любую другую клавишу: ")
print("Работа программы завешена")
```

В данном случае цикл ***while*** будет продолжаться, пока переменная ***choice*** содержит латинскую букву `"Y"` или `"y"`.

Сам блок цикла состоит из двух инструкций. Сначала выводится сообщение `"Привет"`, а потом вводится новое значение для переменной ***choice***. И если пользователь нажмет какую-то другую клавишу, отличную от ***Y***, произойдет выход из цикла, так как условие `choice.lower() == "y"` вернет значение ***False***. Каждый такой проход цикла называется итерацией.

Также обратите внимание, что последняя инструкция

`print("Работа программы завешена")`

не имеет отступов от начала строки, поэтому она не входит в цикл ***while***.

Дугой пример - вычисление факториала:

```
# Программа по вычислению факториала

number = int(input("Введите число: "))
i = 1
factorial = 1
while i <= number:
    factorial *= i
    i += 1
print("Факториал числа", number, "равен", factorial)
```

Здесь вводит с консоли некоторое число, и пока число-счетчик ***i*** не будет больше введенного числа, будет выполняться цикл, в котором происходит умножения числа ***factorial***.

Консольный вывод:

```
Введите число: 6
Факториал числа 6 равен 720
```

Цикл for

Другой тип циклов представляет конструкция ***for***. Цикл ***for*** вызывается для каждого числа в некоторой коллекции чисел. Коллекция чисел создается с помощью функции `range()`. Формальное определение цикла ***for***:

```
for int_var in range(целое_число):
    инструкции
```

После ключевого слова ***for*** идет переменная ***int_var***, которая хранит целые числа (название переменной может быть любое), затем ключевое слово ***in***, вызов функции `range()` и двоеточие.

А со следующей строки располагается блок инструкций цикла, которые также должны иметь отступы от начала строки.

При выполнении цикла Python последовательно получает все числа из коллекции, которая создается функцией ***range***, и сохраняет эти числа в переменной ***int_var***. При первом проходе цикл получает первое число из коллекции, при втором – второе число и так далее, пока не переберет все числа. Когда все числа в коллекции будут перебраны, цикл завершает свою работу.

Рассмотрим на примере вычисления факториала:

```
# Программа по вычислению факториала

number = int(input("Введите число: "))
factorial = 1
for i in range(1, number+1):
    factorial *= i
print("Факториал числа", number, "равен", factorial)
```

Вначале вводим с консоли число. В цикле определяем переменную ***i***, в которую сохраняются числа из коллекции, создаваемой функцией ***range***.

Функция ***range*** здесь принимает два аргумента – начальное число коллекции (здесь число 1) и число, до которого надо добавлять числа (то есть `number + 1`).

Допустим, с консоли вводится число 6, то вызов функции range приобретает следующую форму:

```
range(1, 6 + 1):
```

Эта функция будет создавать коллекцию, которая будет начинаться с 1 и будет последовательно наполняться целыми числами вплоть до 7. То есть это будет коллекция [1, 2, 3, 4, 5, 6].

При выполнении цикла из этой коллекции последовательно будут передаваться числа в переменную ***i***, а в самом цикле будет происходить умножение переменной ***i*** на переменную ***factorial***. В итоге мы получим факториал числа.

Консольный вывод программы:

```
Введите число: 6
Факториал числа 6 равен 720
```

Функция range

Функция ***range*** имеет следующие формы:

- ***range(stop)***: генерирует все целые числа от 0 до stop.
- ***range(start, stop)***: генерирует все целые числа в промежутке от start (включая) до stop (не включая). Выше в программе факториала использована именно эта форма.
- ***range(start, stop, step)***: генерирует целые числа в промежутке от start (включая) до stop (не включая), которые увеличиваются на значение step.

Примеры вызовов функции ***range***:

```
range(5)            # 0, 1, 2, 3, 4
range(1, 5)         # 1, 2, 3, 4
range(2, 10, 2)     # 2, 4, 6, 8
range(5, 0, -1)     # 5, 4, 3, 2, 1
```

Например, выведем последовательно все числа от 0 до 4:

```
for i in range(5):
    print(i, end=" ")
```

Важно: range() **не возвращает список**, а возвращает уникальный объект (*экземпляр класса range*), который обладает свойствами, подобными списку и генератору. Поскольку он действует как последовательность, мы можем получить доступ к его элементам, используя индексы. Он допускает как положительные, так и отрицательные значения индекса.

Подробнее: https://docs.python.org/3.11/library/stdtypes.html#range

Вложенные циклы

Одни циклы внутри себя могут содержать другие циклы. Рассмотрим на примере вывода таблицы умножения:

```
for i in range(1, 10):
    for j in range(1, 10):
        print(i * j, end="\t")
    print("\n")
```

Внешний цикл `for i in range(1, 10)` срабатывает **9 раз**, так как в коллекции, возвращаемой функцией ***range***, 9 чисел. Внутренний цикл `for j in range(1, 10)` срабатывает 9 раз для одной итерации внешнего цикла, и соответственно 81 раз для всех итераций внешнего цикла.

В каждой итерации внутреннего цикла на консоль будет выводиться произведение чисел ***i*** и ***j***.

Выход из цикла. break и continue

Для управления циклом мы можем использовать специальные операторы ***break*** и ***continue***. Оператор ***break*** осуществляет выход из цикла. А оператор ***continue*** выполняет переход к следующей итерации цикла.

Оператор ***break*** может использоваться, если в цикле образуются условия, которые несовместимы с его дальнейшим выполнением.

Рассмотрим следующий пример:

```
# Программа Обменный пункт

print("Для выхода нажмите Y")

while True:
    data = input("Введите сумму для обмена: ")
    if data.lower() == "y":
        break  # выход из цикла
    money = int(data)
    cache = round(money / 65, 2)
    print("К выдаче", cache, "долларов")

print("Работа обменного пункта завершена")
```

Здесь мы имеем дело с бесконечным циклом, так как условие `while True` всегда истинно и всегда будет выполняться. Это популярный прием для создания программ, которые должны выполняться неопределенно долго.

В самом цикле получаем ввод с консоли. Мы предполагаем, что пользователь будет вводить число - условную сумму денег для обмена. Если пользователь вводит букву `"Y"` или `"y"`, то с помощью оператора ***break*** выходим из цикла и прекращаем работу программы. Иначе делим введенную сумму на обменный курс, с помощью функции ***round*** округляем результат и выводим его на консоль. И так до бесконечности, пока пользователь не захочет выйти из программы, нажав на клавишу ***Y***.

Консольный вывод программы:

```
Для выхода нажмите Y
Введите сумму для обмена: 20000
К выдаче 307.69 долларов
Введите сумму для обмена: Y
Работа обменного пункта завершена
```

Но что, если пользователь введет отрицательное число? В этом случае программа также выдаст отрицательный результат, что не является корректным поведением.

И в этом случае перед вычислением мы можем проверить значение, меньше ли оно нуля, и если меньше, с помощью оператора ***continue*** выполнить переход к следующей итерации цикла без его завершения:

```
# Программа Обменный пункт

print("Для выхода нажмите Y")

while True:
    data = input("Введите сумму для обмена: ")
    if data.lower() == "y":
        break  # выход из цикла
    money = int(data)
    if money < 0:
        print("Сумма должна быть положительной!")
        continue
    cache = round(money / 65, 2)
    print("К выдаче", cache, "долларов")

print("Работа обменного пункта завершена")
```

Также обращаю внимание, что для определения, относится ли инструкция к блоку ***while*** или к вложенной конструкции ***if***, опять же используются отступы.

И в этом случае мы уже не сможем получить результат для отрицательной суммы:

```
Для выхода нажмите Y
Введите сумму для обмена: -20000
Сумма должна быть положительной!
Введите сумму для обмена: 20000
К выдаче 307.69 долларов
Введите сумму для обмена: y
Работа обменного пункта завершена
```

Требования к выполнению лабораторной работы №1

1. Изучите теоретическую часть к первой лабораторной работе.
2. Установите Python + IDE.
3. Создайте новый проект.
4. Запустите примеры из лабораторной работы.
5. Выполните задание согласно вашему варианту (*вариант узнавайте у своего преподавателя*).

Формат защиты лабораторных работ:

1. Продемонстрируйте выполненные задания.
2. Ответьте на вопросы по вашему коду.
3. При необходимости выполните дополнительное (*дополнительные*) задания от преподавателя.
4. Ответьте (*устно*) преподавателю на контрольные вопросы.

Список вопросов

1. Для чего необходимо устанавливать различные версии интерпретатора отдельно для каждого проекта?
2. Что такое инструкция?
3. Что означает термин «динамическая типизация»?
4. В чём отличие *list* от *tuple*?
5. В чём отличие *set* от *list*?
6. Что возвращает функция *range*?

Задания

Во всех заданиях необходимо проверять корректность вводимых данных и выводить соответствующие сообщения об ошибках.

Вариант №1

Задание №1. Пользователь с клавиатуры вводит две строки *s* и *x*, где *s* – исходная строка, *x* – «вирус». Необходимо из исходной строки s удалить все вхождения строки *x*, таким образом, чтобы в результирующей строке не осталось «вирусов».

** Строки не чувствительны к регистру.*

Примеры:

s	*x*	*result*
python	py	thon
tuple	Up	tle
Queues	ue	Qs

Задание №2. Пользователь с клавиатуры вводит *n* целых положительных чисел (*через пробел*) – получаем список a_n. Вам доступна следующая операция:

- выберите отрезок $[a_i, a_j]$, такой, что $1 \leqslant i \leqslant j \leqslant n$
- уменьшите элементы $a_i, a_{i+1}, a_{i+2}, \ldots, a_j$ на единицу

Задача вывести число *k* – минимальное количество операций, после которых все элементы списка будут равны нулю.

Примеры:

a	*k*	*Примечание*
[2, 2]	2	[2, 2] => [1, 1] => [0, 0]
[4, 4, 5, 5]	5	[4, 4, 5, 5] => [3, 3, 4, 4] => [2, 2, 3, 3] => [1, 1, 2, 2] => [0, 0, 1, 1] => [0, 0, 0, 0]

Задание №3.

Администратору кинотеатра необходимо вести учёт стоимости билетов. Цена на билет в различные дни/часы может изменяться, поэтому для пары ряд/место, может быть задано несколько строк.

Входные данные:

Пользователь вводит целое положительное число *n* – количество строк.

Затем вводит n строк формата:

{ряд} {место} {стоимость_билета}

Например, строка

1 2 1000

означает: 1-й ряд, 2 место, 1000 руб.

Выходные данные:

m пар *{ряд$_i$} {место$_i$}* – {k_i}

где k_i – количество различных возможных цен билета на *{ряд$_i$} {место$_i$}*

Примеры:

n	*tickets*	*k*	*Примечание*
4	1 1 1000 1 1 1000 1 2 2000 1 2 3000	1 1 – 1 1 2 – 2	Билет на 1-й ряд 1 место во всех кейсах имеет только одну цену (1000 руб.) => $k_1 = 1$ Билет на 1-й ряд 2 место имеет 2 различные цены (2000 руб. и 3000 руб.) => $k_2 = 2$
3	1 1 1000 1 1 2000 1 1 2000	1 1 – 2	Билет на 1-й ряд 1 место имеет две различные цены (1000 руб. и 2000 руб.) => $k_1 = 2$

Примечание: в качестве ключей словаря (*dict*) могут быть использованы кортежи (*tuple*).

Вариант №2

З**адание №1.** Пользователь с клавиатуры вводит строку *s (разрешаются только латинские символы без пробелов)*. Необходимо вывести целое число *n* – количество индексов *i* таких, что после удаления символа s_i из исходной строки *s* новая строка *s'* становится палиндромом. Палиндромом называется строка, которая одинаково читается как слева направо, так и справа налево.

Примеры:

s	*n*	*Примечание*
dad	1	При удалении первого символа: ad – не палиндром При удалении второго символа: dd – **палиндром** При удалении третьего символа: da – не палиндром
qq	2	При удалении первого символа: d – **палиндром** При удалении второго символа: d – **палиндром**
Level	1	evel – не палиндром Lvel – не палиндром Leel – **палиндром** Levl – не палиндром Leve – не палиндром

Задание №2. Пользователь с клавиатуры вводит целые неотрицательные числа (*через пробел*). Необходимо отсортировать список чисел по количеству вхождений единиц в бинарном представлении числа. Если два числа имеют одинаковое количество единиц (*битов*), вместо этого сравните их реальные значения.

Примеры:

list	*result*	*Примечание*
[2, 1, 3]	[1, 2, 3]	1 – 0**1**; 2 – **1**0; 3 – **11** 1 и 2 имеют одинаковое количество «единиц», но так как 2 > 1 => число 1 имеет меньший приоритет.
[8, 16]	[8, 16]	8 – **1**000; 16 – **1**0000
[15, 32, 63, 64]	[32, 64, 15, 63]	32 – 0**1**00000; 64 – **1**000000; 15 – 000**1111**; 63 – 0**111111**;

Задание №3.

Сетевому администратору необходимо проанализировать логи авторизации. Каждая строка такого лога представляет себе данные о дате авторизации пользователя и его IP адресе.

Входные данные:

Пользователь вводит целое положительное число *n* – количество строк.

Затем вводит n строк формата:

{логин} {дата} {IP}

Login – набор латинский букв без пробелов (например, Admin).

Дата – дата формата dd.mm.YYYY (например, 31.12.2021).

IP – IP адрес формата IPv4 (например, 10.0.0.2).

Выходные данные:

Вывести *логин* с максимальным количеством различных IP адресов за один день. Если таких несколько, то вывести любой из них.

Примеры:

n	*Логи*	*Логин*	*Примечание*
5	Admin 01.01.2021 10.0.0.2 Admin 01.01.2021 10.0.0.3 User 01.01.2021 192.168.0.1 User 02.01.2021 192.168.0.2 User 03.01.2021 192.168.0.3	Admin	01.01.2021 пользователь с логином Admin авторизовался с двух различных адресов. Пользователь с логином User хоть и авторизовался с 3-х различных адресов, но все в разные дни.
4	Admin 01.01.2021 10.0.0.2 User 02.01.2021 192.168.0.1 User 02.01.2021 192.168.0.2 User 02.01.2021 192.168.0.3	User	Пользователь User авторизовался с 3-х различных IP адресов за один день (02.01.2021)

Примечание: в качестве ключей словаря (*dict*) могут быть использованы кортежи (*tuple*). Проверять корректность IP адреса и/или даты желательно, но не является обязательной частью задания.