Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» и профилизации "Искусственный интеллект и математическая экономика" ММФ БГУ

Тема 3. Операторы в Python

доц. Лаврова О.А., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

март, 2025

Оператор (statement) — это конструкция языка для указания действий при выполнении программы. Оператор в Python задается с помощью зарезервированных слов языка (for, while, def, class и др.) или символа =. *Оператор не может быть частью выражения.*

Выражение (expression) в Python — это конструкция языка в виде комбинации литералов, переменных, операций и вызовов функций, которая вычисляется и возвращает значение в виде объекта.

В языке Python определены следующие операторы: оператор присваивания `=` и `:=`, оператор выбора `if`, операторы цикла `while` и `for`, `break`, `continue`, оператор сопоставления `match`, оператор-заполнитель `pass`, оператор удаления переменной `del`, оператор определения функции `def`, операторы выхода из функции `return`, `yield`, и `yield from`, операторы объявления `global` и `nonlocal`, оператор импортирования модуля `import`, оператор определения класса `class`, операторы для обработки исключений `assert`, `raise`, `try`, `with`.

3.1 Несоставной и составной оператор

Операторы в Python делятся на несоставные и составные операторы. **Составной оператор** — это оператор, в который вложены другие операторы и выражения. **Несоставной оператор** — это оператор, в который нельзя вложить операторы и выражения. Примеры несоставных операторов: `pass`, `=`, `del`, `import` и др.

Синтаксис несоставного оператора

Несоставные операторы обычно располагают по одному в строке кода. Конец строки *автоматически* завершает несоставной оператор/выражение, находящийся

в этой строке. Символ **точка с запятой** `;` в конце несоставного оператора/выражения НЕ обязателен.

Несколько несоставных операторов и/или выражений можно помещать в одну строку кода, разделяя их символом точка с запятой `;`

In [8]:
```
a = 1; print(a); b = [1]
```

```
1
```

Для читабельности кода рекомендовано размещать по одному несоставному оператору в строке.

Код, содержащий круглые (), квадратные [] или фигурные {} скобки, может занимать несколько строк. **Многострочный оператор со скобками/выражение со скобками** не закончится до тех пор, пока не достигнет закрывающей скобки из пары. Отступы в строках продолжения роли не играют, хотя строки рекомендовано выравнивать ради читабельности

In [11]:
```
a = [(1,
      2),
     {3,
      4}]; a
```

Out[11]:
```
[(1, 2), {3, 4}]
```

Примеры из **PEP 8** (руководство по стандартам стиля кодирования на Python https://www.python.org/dev/peps/pep-0008/) корректного выравнивания многострочного кода при вызове функции:

```
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

При написании многострочного кода следует использовать **круглые скобки** `()`

In [14]:
```
b = (a
     + [5]
     + 2*a); b
```

Out[14]:
```
[(1, 2), {3, 4}, 5, (1, 2), {3, 4}, (1, 2), {3, 4}]
```

Альтернативным синтаксисом для написания многострочного кода является использование символа **обратной косой черты** `\` в конце каждой строки кода

In [16]:
```
b = a + [5] +\
     2*a; b
```

Out[16]:
```
[(1, 2), {3, 4}, 5, (1, 2), {3, 4}, (1, 2), {3, 4}]
```

На текущий момент такой синтаксис использовать не рекомендуется, так как он подвержен ошибкам. Символ обратной косой черты `\` можно не заметить, его можно забыть написать.

Синтаксис составного оператора

Составной оператор — это оператор, в который вложены другие операторы и выражения. Примеры несоставных операторов: `if`, `match`, `for`, `while` и др.

Составной оператор имеет следующий синтаксис:

```
Строка заголовка оператора:

    Тело оператора в виде вложенного блока операторов и выражений
```

In [21]:
```
if 'KM2':
    print('KM2')
```

```
KM2
```

Строка заголовка составного оператора должна заканчиваться символом **двоеточие** `:`. Отсутствие символа **двоеточие** `:` в конце строки заголовка составного оператора является наиболее распространенной ошибкой в Python.

Тело составного оператора записывается с отступом относительно строки заголовка оператора. **Отступ** — это пробельные символы (пустое пространство) слева от кода. Операторы и выражения вложенного блока должны быть смещены на *одинаковое расстояние* вправо:

In [24]:
```
if 'KM2':
    print('KM2')
    print('Body ends here')
```

```
KM2
Body ends here
```

Для обозначения вложенного блока оператора используется только отступ, скобки не нужны.

Для отступа можно использовать либо *пробелы*, либо *табуляции*. В отдельном блоке используйте либо *пробелы*, либо *табуляции*, но не то и другое одновременно. Общепринято использовать отступ в *4 пробела* или *1 табуляцию*.

Делайте бОльший отступ вправо для следующего вложенного блока и меньший отступ для закрытия предыдущего блока. На основании отступов будет *автоматически* определяться начало и конец вложенного блока:

In [28]:
```
if 'KM2':
    print('\t outer if')
    if True:
        print('\t \t inner if')
```

```
    print('\t outer if')
print('outside')
```

```
        outer if
                inner if
        outer if
outside
```

Выравнивание кода в соответствии с его структурой с применением отступов является частью синтаксиса языка Python и является *особенностью языка Python*. Совокупный эффект заключается в том, что код на Python становится более согласованным и читабельным.

Старайтесь выравнивать код в ЛЮБОМ блочно-структурированном языке программирования для отражения логической структуры кода в целях читабельности кода.

Пробельные символы внутри операторов и выражений почти всегда игнорируются. Исключением являются ситуации, когда пробелы находятся внутри строковых литералов и когда пробелы применяются для отступа в составных операторах.

In [32]:
```
a=1+2; a = 1 + 2; '1+2', '1 + 2'
```

Out[32]:
```
('1+2', '1 + 2')
```

Если тело составного оператора состоит только из выражений или несоставных операторов, то тело составного оператора может быть записано в строке заголовка оператора после символа двоеточие:

In [34]:
```
if True: print(True + 2)
```

```
3
```

Рекомендации по оформлению кода

PEP 8 — руководство по стандартам стиля кодирования на Python, см. https://www.python.org/dev/peps/pep-0008/ (2001, изменения в 2013)

Перевод **PEP 8**, см. например, https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html#section-2

3.2 Оператор присваивания `=`

Явное присваивание

Явное присваивание в Python выполняется с помощью оператора присваивания `=`. Слева от оператора присваивания указывается имя переменной, справа — выражение. В процессе выполнения явного присваивания сначала вычисляется значение выражения, затем имя переменной *связываются* с результирующим объектом.

После выполнения оператора присваивания `=` переменная хранит ссылку на объект: **переменная -> объект**.

Синтаксические формы явных присваиваний:

1. базовое присваивание
2. присваивание последовательностей
3. расширенное присваивание последовательностей
4. групповое присваивание
5. дополненное присваивание

1. Базовое присваивание

Базовое присваивание связывает *одну переменную*, стоящую слева от оператора присваивания `=`, с *одним объектом*, представляющим результат вычисления выражения, стоящего справа от оператора присваивания

In [45]:
```
variable = 'object'.capitalize()
variable
```

Out[45]:
```
'Object'
```

2. Присваивание последовательностей

Оператор присваивания последовательностей имеет вид:

`список или кортеж переменных (имен) = последовательность`

и связывает переменные, указанные в его левой части, с элементами последовательности, которая указана в правой части.

Оператор присваивания последовательностей является *позиционным оператором*. У последовательностей, стоящих в левой и правой частях, связываются элементы, стоящие на одинаковых позициях. В результате для каждого `i`-го элемента последовательности определена ссылка на объект:

`переменная(имя)i -> объект(значение выражения)i`.

Следует отметить, что количество элементов последовательностей левой и правой частей должно совпадать.

In [49]:
```
a, b = 1, 2
a, b
```

Out[49]:
```
(1, 2)
```

Обратите внимание, что присваивание переменных осуществляется *последовательно* слева направо. Это приводит к влиянию переменных, стоящих

левее текущего присваивания в последовательности, на результаты присваиваний для переменных, стоящих правее текущего присваивания

In [51]:
```
x = list(range(3))
i = 0
x[i], i, x[i] = 'first', 2, 'last'
x
```

Out[51]:
```
['first', 1, 'last']
```

Распространенный кодовый трюк в Python для обмена значениями двух переменных без использования дополнительной переменной на основе присваивания последовательностей

In [53]:
```
print(a, b)
a, b = b, a
a, b
```

```
1 2
```

Out[53]:
```
(2, 1)
```

Слева от оператора присваивания может стоять кортеж или список

`var1, var2, ..., varN = obj1, obj2, ..., objN`

`[var1, var2, ..., varN] = obj1, obj2, ..., objN`

В правой части оператора присваивания может стоять не только последовательность, но и итерируемый объект:

In [55]:
```
a, b = {11:1, 12:2}; [c,d] = range(2)
a, b, c, d
```

Out[55]:
```
(11, 12, 0, 1)
```

3. Расширенное присваивание с использованием *

Слева от оператора присваивания можно использовать одиночную переменную, отмеченную звездочкой, `*var` для указания расширенного присваивания последовательностей. В этом случае переменной `var`, отмеченной звездочкой, присваивается *список*, в котором собраны все элементы последовательности, не присвоенные остальным переменным

In [58]:
```
a, *b = 'object'
a, b
```

Out[58]:
```
('o', ['b', 'j', 'e', 'c', 't'])
```

Если нет объектов, которые можно было бы сопоставить с переменной, отмеченной звездочкой, тогда ей присваивается пустой список:

In [60]:
```
a, *b = "a"
a, b
```

Out[60]:
```
('a', [])
```

Кодовый трюк в Python на основе расширенного присваивания последовательностей для получения первого и остальных объектов последовательности или начальных и последнего объектов последовательности

In [62]:
```
first, *rest = 'object'
*most, last = 'object'
first, rest, most, last
```

Out[62]:
```
('o', ['b', 'j', 'e', 'c', 't'], ['o', 'b', 'j', 'e', 'c'], 't')
```

4. Групповое присваивание

При групповом присваивании `var1 = var2 = ... = varN = obj` переменные `var1`, `var2`, ..., `varN` ссылаются на один и тот же объект `obj`.

In [65]:
```
a = b = [1]
id(a), id(b)
```

Out[65]:
```
(1517389223104, 1517389223104)
```

Для сравнения: при присваивании последовательностей переменные ссылаются на разные объекты

In [67]:
```
a, b = [1], [1]
id(a), id(b)
```

Out[67]:
```
(1517389315776, 1517389311424)
```

5. Дополненное присваивание

Оператор дополненного присваивания имеет вид:

`a operation= b`

и указывает следующую последовательность действий: сначала вычисляется выражение `a operation b`, затем полученный объект(значение выражения) связывается с переменной `a`, т.е. `a -> a operation b`. Всего существует 12 операторов дополненного присваивания, например, `+=`, `/=`, `*=` и т.д.

In [70]:
```
a = b = 1
a += 1;  b *= 3
a, b
```

Out[70]:
```
(2, 3)
```

Связывание имени переменной с пространством имен

В Python важным является место в программе, где переменной присваивается значение.

Местоположение присваивания переменной ее значения, например, `x = value`, ассоциирует или связывает переменную `x` с пространством имен, в котором переменная будет находиться.

При присваивании вида `obj.x = value` создается или изменяется переменная `x` в **пространстве имен объекта** `obj`.

Переменная, которой выполнено присваивание внутри функции (внутри оператора `def`), ассоциируется или связывается с **пространством имен функции**.

Переменная, которой выполнено присваивание внутри класса (внутри оператора `class`), ассоциируется или связывается с **пространством имен класса**.

Переменная, которой выполнено присваивание в модуле (не внутри функции или класса), ассоциируется с **пространством имен модуля**.

Даже если при присваивании переменные задаются одинаковыми именами, присваивание значений в разных местах программы (внутри модуля, внутри функции, внутри класса, внутри метода класса) делает все переменные уникальными

In [75]:
```
x = 1 # пространство имен модуля

def f():
    x = 10 # пространство имен функции
    print(f'{x=}')

f()
print(f'{x=}')
```

```
x=10
x=1
```

Вызов встроенной функции `dir()` без аргументов позволяет просмотреть пространство имен, соответствующее местоположению вызова функции `dir()`

In [77]:
```
x = 1 # пространство имен модуля
print(dir())

def f(e=2):
    x = 10 # пространство имен функции
    print(dir())

f()
```

```
['In', 'NamespaceMagics', 'Out', '_', '_1', '_10', '_11', '_12', '_13', '_14', '_
15', '_16', '_17', '_18', '_19', '_2', '_20', '_22', '_23', '_25', '_26', '_27',
'_29', '_3', '_30', '_31', '_32', '_33', '_35', '_36', '_37', '_38', '_39', '_4',
'_40', '_41', '_42', '_43', '_44', '_45', '_46', '_47', '_48', '_49', '_5', '_5
0', '_51', '_52', '_53', '_54', '_55', '_56', '_57', '_58', '_59', '_6', '_60',
'_61', '_62', '_63', '_64', '_65', '_66', '_67', '_68', '_69', '_7', '_70', '_7
1', '_72', '_73', '_74', '_76', '_9', '__', '___K', '____', '__builtin__', '__built
ins__', '__doc__', '__import__', '__ipywidgets', '__loader__', '__name__', '__n
p', '__package__', '__pandas', '__pd', '__pyspark', '__session__', '__spec__', '_
_tf', '__torch', '__xr', '_attempt_import', '_check_imported', '_dh', '_i', '_i
1', '_i10', '_i11', '_i12', '_i13', '_i14', '_i15', '_i16', '_i17', '_i18', '_i1
9', '_i2', '_i20', '_i21', '_i22', '_i23', '_i24', '_i25', '_i26', '_i27', '_i2
8', '_i29', '_i3', '_i30', '_i31', '_i32', '_i33', '_i34', '_i35', '_i36', '_i3
7', '_i38', '_i39', '_i4', '_i40', '_i41', '_i42', '_i43', '_i44', '_i45', '_i4
6', '_i47', '_i48', '_i49', '_i5', '_i50', '_i51', '_i52', '_i53', '_i54', '_i5
5', '_i56', '_i57', '_i58', '_i59', '_i6', '_i60', '_i61', '_i62', '_i63', '_i6
4', '_i65', '_i66', '_i67', '_i68', '_i69', '_i7', '_i70', '_i71', '_i72', '_i7
3', '_i74', '_i75', '_i76', '_i77', '_i8', '_i9', '_ih', '_ii', '_iii', '_jupyter
lab_variableinspector_Jupyter', '_jupyterlab_variableinspector_default', '_jupyte
rlab_variableinspector_deletevariable', '_jupyterlab_variableinspector_dict_lis
t', '_jupyterlab_variableinspector_displaywidget', '_jupyterlab_variableinspector
_getcontentof', '_jupyterlab_variableinspector_getmatrixcontent', '_jupyterlab_va
riableinspector_getshapeof', '_jupyterlab_variableinspector_getsizeof', '_jupyter
lab_variableinspector_is_matrix', '_jupyterlab_variableinspector_is_widget', '_ju
pyterlab_variableinspector_nms', '_oh', 'a', 'b', 'c', 'd', 'dataframe_columns',
'dataframe_hash', 'dtypes_str', 'exit', 'f', 'first', 'get_dataframes', 'get_ipyt
hon', 'getpass', 'hashlib', 'i', 'import_pandas_safely', 'is_data_frame', 'json',
'last', 'most', 'open', 'quit', 'rest', 'sys', 'variable', 'x']
['e', 'x']
```

Неявное присваивание

При неявном присваивании переменная создается *автоматически* не при выполнении оператора присваивания `=`, а в процессе выполнения других действий.

Возможные ситуации:

- при организации циклов `for` происходит *неявное присваивание* переменной цикла значения из итерируемого объекта
- при определении функции происходит *неявное присваивание* имени функции объекта-функции (оператор `def`, Тема 4)
- при вызове функции происходит *неявное присваивание* переменным-аргументам значений-объектов (Тема 4)
- при импортировании модуля происходит *неявное присваивание* имени модуля объекта-модуля (оператор `import`, Тема 7)
- при определении класса происходит *неявное присваивание* имени класса объекта-класса (оператор `class`, Тема 8)

3.3 Оператор выбора `if`

Оператор `if` используется для выбора варианта кода на основе результата проверки условия.

Оператор `if` является составным и имеет следующий шаблон:

```
if условие 1:
        Вложенный блок if
elif условие 2: # необязательная конструкция; может быть несколько
        Вложенный блок elif
else:           # необязательная конструкция
        Вложенный блок else
```

Конструкции `elif` и `else` являются частью оператора `if`. Конструкции `elif` и `else` имеют собственные строки заголовков, а также собственные вложенные блоки кода.

In [85]:
```
test = 1
if test<5:
    print(f'{test} is less than 5')
elif test==5:
    print(f'{test} equals 5')
else:
    print(f'{test} is more than 5')
```

```
1 is less than 5
```

Строки заголовков оператора `if` и конструкций `elif` и `else` связываются друг с другом выравниванием с одинаковыми отступами.

Тернарное выражение `if-else`

In [88]:
```
test = 14
answer = f'{test} < 5' if test<5 else f'{test} >= 5'
answer
```

Out[88]:
```
'14 >= 5'
```

Тернарное выражение `if` может быть использовано внутри других выражений:

In [90]:
```
answer = 100 + (test if test<10 else 0)
answer
```

Out[90]:
```
100
```

Булевые операции

Для построения условных выражений можно использовать булевые операции `and`, `or`, `not`:

In [93]:
```
1<3 and 3<5, 1<3<5, True or False, not 1<3
```

```
Out[93]: (True, True, True, False)
```

При выполнении операции `and` вычисляются слева направо булевые значения операндов и возвращается *первый ложный объект*, если он найден, или последний операнд в противном случае. Выполнение операции останавливается на первом найденном ложном объекте. Это называют **укороченной оценкой**, так как вычисления для оставшегося выражения прекращается, как только результат становится известным.

```
In [95]: bool([]), bool({}), [] and {}  # and вернула первый ложный операнд
```

```
Out[95]: (False, False, [])
```

```
In [96]: bool(1), bool(3), 1 and 3  # and вернула последний операнд
```

```
Out[96]: (True, True, 3)
```

При выполнении операции `or` вычисляются слева направо булевые значения операндов и возвращается *первый истинный объект*, если он найден, или последний операнд в противном случае. Выполнение операции останавливается на первом найденном истинном объекте. Это называют *укороченной оценкой*.

```
In [98]: bool(1), bool(3), 1 or 3 # or вернула первый истинный операнд
```

```
Out[98]: (True, True, 1)
```

```
In [99]: bool([]), bool({}), [] or {} # or вернула последний операнд
```

```
Out[99]: (False, False, {})
```

3.4 Операторы цикла `for` и `while`

В Python определены два оператора цикла: `for` и `while`.

Цикл на основе оператора `for` является простым и эффективным способом *прохода по элементам* итерируемого объекта и выполнения блока кода для каждого элемента по очереди.

Цикл на основе оператора `while` представляет собой наиболее универсальный инструмент для организации циклов *на основе выполнения условия*; он не ограничивается проходом по итерируемому объекту, но в целом требует написания большего объема кода и выполняется медленнее, чем цикл `for`.

В Python существуют также *неявные* концепции организации циклов:

- списковое включение и его аналоги для словарей, множеств и генераторных выражений
- операция `in` проверки принадлежности
- функции `map`, `filter`, `reduce`

- генераторные функции

Включения выполняются со скоростью кода на языке C внутри интерпретатора, которая гораздо выше, чем скорость выполнения байт-кода циклов `for` и `while` внутри PVM.

Оператор `while`

Оператор `while` является составным и имеет следующий шаблон:

```
while условие:

        Вложенный блок while

else: # необязательная конструкция

        Вложенный блок else

while условие:

        Вложенный блок while

else: # необязательная конструкция

        Вложенный блок else
```

Вложенный блок while может содержать несоставные операторы `break` и `continue`. Оператор `break` завершает выполнение всей структуры повторения и осуществляет переход за пределы заключающего оператора цикла. Оператор `continue` прерывает выполнение текущей итерации цикла и осуществляет переход на выполнение следующей итерации этого цикла. Несоставные операторы `break` и `continue` используются только в циклах.

```
while условие:

        Вложенный блок while

else: # необязательная конструкция

        Вложенный блок else
```

Вложенный блок else выполняется один раз и только в том случае, если выход из цикла произошел при невыполнении условия в строке заголовка. Другими словами, *Вложенный блок else* выполняется только в том случае, когда *Вложенный блок while* НЕ закончился выполнением оператора `break`.

Совместное использование `while` и `break` пропускает *Вложенный блок else* конструкции `else`, что предлагает структурированный способ обработки ситуации с неудавшимся поиском в цикле

In [111…

```
test_seq = range(10)
control = 9
while test_seq:
    first, *rest = test_seq
    if first==control:
        print(f'{control} is found')
        break
    test_seq = rest
else:
    print(f'{control} is not found')
```

```
9 is found
```

Конструкция `else` цикла `while` является *особенностью* Python и не поддерживается в других языках программирования.

Оператор `for`

Оператор `for` является составным и имеет следующий шаблон:

```
for element in iterable:

        Вложенный блок for

else: # необязательная конструкция

        Вложенный блок else

for element in iterable:

        Вложенный блок for

else: # необязательная конструкция

        Вложенный блок else
```

Переменной цикла `element` последовательно *неявно присваиваются* объекты из итерируемого объекта `iterable` слева направо, пока не будет достигнут конец итерируемого объекта.

```
for element in iterable:

        Вложенный блок for

else: # необязательная конструкция

        Вложенный блок else
```

Вложенный блок for может содержать несоставные операторы `break` и `continue`. Оператор `break` завершает выполнение всей структуры повторения и осуществляет переход за пределы заключающего оператора цикла. Оператор `continue` прерывает выполнение текущей итерации цикла и осуществляет

переход на выполнение следующей итерации этого цикла. Операторы `break` и `continue` используются только в циклах.

```
for element in iterable:

        Вложенный блок for

else: # необязательная конструкция

        Вложенный блок else
```

Вложенный блок else выполняется один раз и только в том случае, если выход из цикла произошел после прохода по всему итерируемому объекту `iterable`. Другими словами, *Вложенный блок else* выполняется только в том случае, когда *Вложенный блок for* НЕ закончился выполнением оператора `break`.

Совместное использование `for` и `break` пропускает *Вложенный блок else* конструкции `else`, что предлагает структурированный способ обработки ситуации с неудавшимся поиском в цикле

In [122…
```
control = 10
for i in range(10):
    if i==control:
        print(f'{control} is found')
        break
else:
    print(f'{control} is not found')
```

```
10 is not found
```

Конструкция `else` цикла `for` является *особенностью* Python и не поддерживается в других языках программирования.

Проход по КЛЮЧАМ словаря с помощью цикла `for` можно осуществить двумя способами, когда итерируемым объектом для прохода в цикле является словарь либо результат вызова словарного метода `keys`

In [125…
```
dict1 = dict(enumerate('abcdefg'))

for i in dict1:
    print(i, end=' ')

for i in dict1.keys():
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 0 1 2 3 4 5 6
```

Для прохода по ЗНАЧЕНИЯМ словаря с помощью цикла `for` необходимо в качестве итерируемого объекта для прохода в цикле использовать результат вызова словарного метода `values`:

In [127…
```
print(dict1)
```

```
for i in dict1.values():
    print(i, end=' ')
```

```
{0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g'}
a b c d e f g
```

Для одновременного прохода по КЛЮЧАМ и ЗНАЧЕНИЯМ словаря с помощью цикла `for` необходимо в качестве итерируемого объекта для прохода в цикле использовать результат вызова словарного метода `items`:

In [129…
```
for (key, elem) in dict1.items():
    print(f'{key} => {elem}', end=', ')
```

```
0 => a, 1 => b, 2 => c, 3 => d, 4 => e, 5 => f, 6 => g,
```

Python предоставляет возможность прохода по строкам файла в цикле `for` для файлового объекта, не вызывая метод построкового чтения `readline`

In [131…
```
with open("tmp.txt") as file:
    for x in file:
        print(x, end='')
```

```
the first line
the second line
the last line
100
200
300
100
200
300
```

Совет 1: всякий раз, когда это возможно, применяйте цикл `for` вместо цикла `while`.

Совет 2: не используйте вызовы `range` в циклах `for` кроме крайних случаев.

Замечание: не используйте оператор присваивания `=` в заголовке циклов `while`.

3.5 Оператор присваивания в выражении `:=` (Walrus operator)

Оператор `:=` появился в версии Python 3.8. Синтаксический шаблон:

```
variable := expression
```

Оператор `:=` выполняет действие, аналогичное вызову оператора `=`, но дополнительно возвращает результирующий объект вычисления выражения `expression`, стоящего справа от оператора. Это позволяет использовать оператор `:=` внутри любого выражения, в отличие от всех других операторов Python.

In [135…
```
(value := 2 ** 3) * 4 + 1, value
```

Out[135…
```
(33, 8)
```

Оператор `:=` может использоваться для исключения повторных вычислений во включениях, в операторе выбора, в циклах.

In [137…
```
%timeit {2**i**i for i in range(5) if 2**i**i%3}
```

```
2.96 µs ± 1.11 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [138…
```
%timeit {value for i in range(5) if (value:=2**i**i)%3}
```

```
2.63 µs ± 49.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Предположим, что необходимо вычислить приближенное значение для числа π с заданной точностью ϵ, используя ряд:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

In [141…
```
import math

eps = 1E-4
n = 0
sum = 0.
while 4*abs(value := ((-1)** n) / (2 * n + 1)) >= eps:
    sum += value
    n += 1
pi = sum*4
pi, abs(pi - math.pi)
```

Out[141…
```
(3.1415426535898248, 4.999999996835314e-05)
```

Каждый член ряда дважды используется в коде: для проверки точности и для вычисления суммы. Использование оператора `:=` позволяет исключить повторное вычисление члена ряда.

3.6 Оператор сопоставления `match`

Оператор сопоставлений `match` появился в версии Python 3.10. Оператор `match` используется для выбора варианта кода на основе сопоставления объекта, указанного в заголовке оператора, с *шаблонами*, указанными в конструкциях `case`.

```
match obj:

    case pattern i: # может быть несколько

                Вложенный блок i
```

Объект `obj` последовательно сравнивается с шаблонами `pattern 1`, `pattern 2` до `pattern N`. При соответствии объекта `obj` шаблону `pattern i` выполняется соответствующий *Вложенный блок i* и действие оператора `match` завершается.

Шаблону `_` соответствует ЛЮБОЙ объект. Часто такой шаблон указывается в последней конструкции `case` в виде `case _:` для обработки ситуации несовпадения объекта `obj` со всеми шаблонами, указанными выше в текущем операторе.

Ознакомьтесь на Примерах 1-7 с результатами выполнения оператора `match` для различных способов задания шаблонов.

В Примере 1 шаблон задается литералом объекта, альтернативными объектами с использованием символа `|`; завершающий шаблон задается символом `_`:

In [148…
```
# Пример 1
language = 'English'
match language:
    case 'russian':
        print('Привет')
    case 'american english' | 'british english' | 'english':
        print('Hello')
    case _:
        print('Undefined')
```

```
Undefined
```

В Примере 2 шаблон задается литералами списков с использованием символа `_`:

In [150…
```
# Пример 2
data = [2,3]
match data:
    case [_]:
        print('one element')
    case [_,_]:
        print('two elements')
```

```
two elements
```

В Примере 3 шаблон задается переменной:

In [152…
```
# Пример 3
language = 'english'
match language:
    case 'russian':
        print('Привет')
    case name:
        print(f'Hello in {name}')
name
```

```
Hello in english
```

Out[152…
```
'english'
```

Дополнительно для переменной в шаблоне можно осуществлять проверку типа объекта

In [154…
```
# Пример 4
language = '5'
match language:
    case 'russian':
```

```
        print('Привет')
    case str(name):
        print(f'Hello in {name}')
    case _:
        print('Undefined language')
```

```
Hello in 5
```

В случае, если значение `language` не совпадает с `'russian'`, тогда происходит неявное присваивание `name=language` и выполняется соответствующий вложенный блок `print(f'Hello in {name}')`. При этом значение переменной `name` сохраняется после выхода из оператора `match`.

В примере 5 шаблон задается словарем с заданными ключами и с переменными в качестве значений

In [157…
```
# Пример 5
book = {'author': 'Бажов', 'title': 'Сказки',
        'year': 1987, 'price': 100}
price = None
match book:
    case {'price': price, 'title': title}:
        print('Incomplete dict works as pattern!')
    case {'author': author, 'title': title, 'year': year}:
        ...
    case _:
        print('Unknown book format')
price
```

```
Incomplete dict works as pattern!
```

Out[157… 100

В примере 6 в шаблоне с переменными используется условие `if`

In [159…
```
# Пример 6
book = {'author': 'Бажов', 'title': 'Сказки',
        'year': 1987, 'price': 100}
match book:
    case {'author': author, 'title': title,
        'year': year, 'price': price} if price<100:
        ...
    case _:
        print('Unknown book format')
```

```
Unknown book format
```

В Примере 7 шаблон задается перечислением переменных или перечислением переменных и переменной со * для реализации распаковки списка или кортежа, указанного в заголовке оператора `match` с последующим присваиванием переменным без * соответствующих элементов последовательности и присваиванием переменной со * списка с оставшимися элементами последовательности:

In [161…
```
# Пример 7
location = [1, 3, 2, "a", "b", "c"]
match location:
```

```
    case x, :
        print(f'1D location found: ({x})')
    case x, y:
        print(f'2D location found: ({x}, {y})')
    case x, y, z, *names:
        print((f'3D location found: ({x}, {y}, {z})'))
        print(f'Also, there is some extra data: {names}')
```

```
3D location found: (1, 3, 2)
Also, there is some extra data: ['a', 'b', 'c']
```

Если при распаковке последовательности мы хотим проигнорировать оставшиеся элементы списка, то можно использовать шаблон вида `*_`: `case x, y, z, *_:`

3.7 Оператор удаления переменной `del`

Оператор `del` удаляет переменную, созданную ранее явным или неявным присваиванием. Удаление переменной означает как удаление имени переменной, так и удаление ссылки на объект, с которым переменная была связана.

In [165…
```
a = 1; del a; a
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[165], line 1
----> 1 a = 1; del a; a

NameError: name 'a' is not defined
```

In [166…
```
list1 = list(range(10))
print(f'{list1 = }')
del list1[0]
print(f'del list1[0]: {list1}')
del list1[1] # будьте внимательны при удалении элементов списка
print(f'del list1[1]: {list1}')
```

```
list1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del list1[0]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
del list1[1]: [1, 3, 4, 5, 6, 7, 8, 9]
```

При удалении переменной объект, на который ссылается переменная, не удаляется. Объект удаляется только в том случае, если при удалении переменной удаляется последняя ссылка на объект. Это приводит к *автоматической* очистке области памяти, где располагается этот объект, так как ни одна переменная в коде на него больше не ссылается.

3.8 Оператор-заполнитель `pass`

Оператор `pass` не выполняет действий и называется оператором-заполнителем. Он часто применяется для создания пустого вложенного блока составного оператора.

In [171…
```
if True:
    pass
```

Оператор `pass` используется для обозначения места в коде, подлежащего заполнению в будущем.

In [174…
```
def f():
    pass
```

В Python 3.X в любом месте кода, где может находиться оператор/выражение/объект, разрешено указывать символ **многоточие** `...`. Многоточие синтаксически заменяет оператор-заполнитель `pass` и объект-заполнитель `None`

In [177…
```
def f():
    x = ...
    ...
```