Тема 2.3 ОСНОВНЫЕ АЛГОРИТМИЧЕСКИЕ ИНСТРУКЦИИ ЯЗЫКА PYTHON

Основные структуры алгоритмов (ОСА) – это определенный набор блоков и стандартных способов их соединения для выполнения типичных последовательностей действий.

Любой алгоритм может быть записан с помощью трёх алгоритмических конструкций: последовательного выполнения команд (линейных алгоритмов), условных операторов и циклов.

Важно знать, что в языке Python синтаксис обладает следующей особенностью: дело в том, что в коде нет операторных скобок (begin..end или {...}); вместо них **отступы** указывают, какие операторы выполнять внутри той или иной конструкции.

1. Последовательность действий (линейные алгоритмы)

Алгоритм называется ***линейным***, если он содержит N шагов, и все шаги выполняются последовательно друг за другом от начала до конца.

Для реализации алгоритмов линейной структуры используются следующие операторы:

1) оператор **input() -** осуществляет ввод данных;
2) оператор **print()** - осуществляет вывод данных;
3) оператор **присваивания (=) -** устанавливает связь между данными и переменными.

Последовательные действия описываются последовательными строками программы. Все операторы, входящие в последовательность действий, должны иметь один и тот же отступ. Например:

```
a = 1
b = 2
a = a + b
b = a - b
a = a - b
print (a, b)
```

2. Оператор условия и выбора (алгоритмы ветвления)

Алгоритм называется ***разветвляющимся***, если последовательность выполнения шагов алгоритма изменяется в зависимости от выполнения некоторых условий. Условие - это логическое выражение, которое может принимать одно из двух значений: True - если условие верно (истинно), и False - если условие неверно (ложно).

В условиях используют знаки отношений: < (меньше), > (больше), <= (меньше или равно), >= (больше или равно), == (равно) и != (не равно).

В качестве условия в условном операторе можно указать любое логическое выражение, в том числе сложное условие, составленное из простых отношений с помощью логических операций (связок) «И», «ИЛИ» и «НЕ» (and, or и not**).**

Операторы сравнения в Python можно объединять в цепочки (в отличие от большинства других языков программирования, где для этого нужно использовать логические связки), например, a == b == c или 1 <= x <= 10.

Разветвляющийся алгоритм можно реализовать в программах с помощью простого, сокращенного, составного операторов, а также конструкции многозначных ветвлений.

Условная инструкция в Питоне имеет следующий синтаксис:

if *Условие****:***
    Блок инструкций 1
else*:*
    Блок инструкций 2

Блок инструкций 1 будет выполнен, если *Условие* истинно. Если *Условие* ложно, будет выполнен *Блок инструкций 2*.

Обратите внимание, что слова **if** и **else** начинаются на одном уровне, а все команды внутренних блоков сдвинуты относительно этого уровня вправо на одно и то же расстояние (4 отступа).

В условной инструкции может отсутствовать слово **else** и последующий блок. Такая инструкция называется ***неполным ветвлением***.

Внутри условного оператора могут находиться любые операторы, в том числе и другие условные операторы. Получаем ***вложенное ветвление*** – после одной развилки в ходе исполнения программы появляется другая развилка. При этом вложенные блоки имеют больший размер отступа (например, 8 пробелов).

if *Условие 1****:***
    Блок инструкций 1
else*:*
    if *Условие 2****:***
        Блок инструкций 2
else*:*
    Блок инструкций 3

Если нужно последовательно проверить несколько условий, используется форма с дополнительным оператором **elif** (сокращение от **else if**) **-** ***оператор выбора***:

if *условие 1:*
    Блок инструкций 1
elif *условие 2:*
    Блок инструкций 2
else*:*
    Блок инструкций 3

Дополнительных условий и связанных с ними блоков **elif** может быть сколько угодно, но важно отметить, что в такой сложной конструкции будет выполнен всегда только один блок кода. Другими словами, как только некоторое условие оказалось истинным, соответствующий блок кода выполняется, и дальнейшие условия не проверяются.

3. Циклы

Циклы - это инструкции, выполняющие одну и ту же последовательность действий многократно.

В Python имеются два вида циклов: **цикл ПОКА** (выполняется некоторое условие) и **цикл ДЛЯ** (всех значений последовательности).

Цикл с условием (while)

Цикл while (“пока”) позволяет выполнить одну и ту же последовательность действий, пока проверяемое условие истинно.

Синтаксис цикла while в простейшем случае выглядит так:

***while** условие:*

блок инструкции (тело цикла)

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. В языке Python тело цикла выделяется отступом.

Один шаг цикла (выполнение тела цикла) называют **итерацией**. Используют цикл **while** всегда, когда какая-то часть кода должна выполниться несколько раз, причем невозможно заранее сказать, сколько именно.

Цикл с переменной (for)

Цикл for, также называемый циклом с параметром, представляет собой цикл обхода заданного множества элементов и выполнения в своем теле различных операций над ними. Множество значений может быть задано списком, кортежем, строкой или диапазоном.

Как правило, циклы **for** используются либо для повторения какой-либо последовательности действий заданное число раз, либо для изменения значения переменной в цикле от некоторого начального значения до некоторого конечного.

Для повторения цикла некоторое заданное число раз можно использовать цикл **for** вместе с функцией **range, с**интаксис:

for ... in range(...):

блок кода (тело цикла)

Range означает «диапазон», то есть for i in range(n) читается как «для (всех) i в диапазоне от 0 (включительно) до n (не включительно)...». Цикл выполняется n раз.

В скобках после слова **range** можно записать не одно, а два или три числа. Эти числа будут интерпретироваться как начальное значение итератора, конечное и его шаг (может быть отрицательным).

Если для **range** задано одно число, то итератор идет от 0 до заданного значения (не включая его).

Если задано два числа, то это начальное значение итератора и конечное, причем указанное конечное значение не входит в диапазон.

Если задано три числа, то это не только начальное и конечное значение итератора, но и шаг итератора. Например:

```
for i in range(4):              # равносильно инструкции for i in 0, 1, 2, 3:
    print(i)                    # выведет на отдельных строчках числа от 0до 3

for i in range(1, 11):
    print(i)                    # выведет на отдельных строчках числа от 1до 10

for i in range(1, 11, 2):
    print(i)                    # выведет на отдельных строчках 1, 3, 5, 7, 9
```

for i in range(10, 0, -1):
print(i) *# выведет на отдельных строчках 10, 9, ..., 1*

Если телом цикла является циклическая структура, то такие циклы называются ***вложенными***. Цикл, содержащий в себе другой цикл, называют *внешним*, а цикл, содержащийся в теле другого цикла, называют *внутренним*.

Синтаксис операторов сложного цикла:

for i in range(Ni, N2): *# внешний цикл*
for j in range(Mi, M2): *# внутренний цикл*
тело цикла

Досрочное завершение цикла

Ходом выполнения цикла можно управлять, для этого применяются операторы **break** (прервать) и **continue** (продолжить) – рисунок 1.

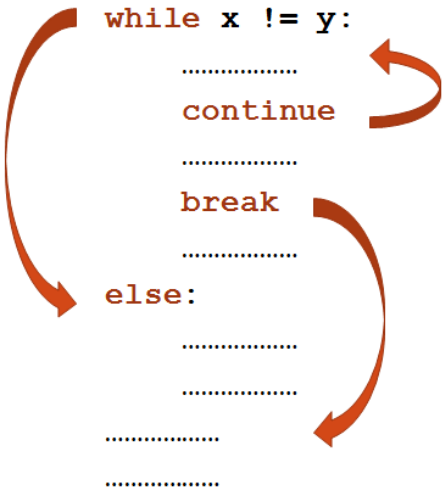

Рисунок 1 - Операторы break и continue

Оператор break прерывает выполнение цикла, управление передается операторам, следующим за оператором цикла.

Break - завершает выполнение цикла на определенном участке кода.

```
for i in 'a b v':
    if i=='b':
        break
    print(i*3)
```

Результат:

aaa

Оператор continue прерывает выполнение очередного шага цикла и возвращает управление в начало цикла, начиная следующий шаг.

Continue - пропускает определенный участок кода.

```
for i in 'a b v':
    if i=='b':
        continue
    print(i*3)
```

Результат:

aaa
vvv

Цикл, который начинается с заголовка **while True** будет выполняться бесконечно, потому что условие True всегда истинно. Выйти из такого цикла можно только с помощью оператора break.

```
while True:
    print ( "Введите положительное число:" )
    n = int ( input() )
    if n > 0:
      break
```

В данном случае он сработает тогда, когда станет истинным условие n > 0, то есть тогда, когда пользователь введет допустимое значение.

4. Исключения. Обработка исключений

Исключения (exceptions) - ещё один тип данных в Python. Часто в работе программы возникают ошибки, препятствующие её дальнейшему выполнению.

Для обработки особых ситуаций (таких как деление на ноль или попытка чтения из несуществующего файла) применяется механизм исключений. Исключения можно обрабатывать, для чего используется конструкция **try–except.**

Лучше всего пояснить синтаксис оператора try-except следующим примером:

```
try:
  res = int(open('a.txt').read()) / int(open('c.txt').read())
  print res
except IOError:
  print "Ошибка ввода-вывода"
except ZeroDivisionError:
  print "Деление на 0"
except KeyboardInterrupt:
  print "Прерывание с клавиатуры"
except:
  print "Ошибка"
```

В этом примере берутся числа из двух файлов и делятся одно на другое. В результате этих действий может возникнуть несколько исключительных ситуаций, некоторые из них отмечены в частях **except** (здесь использованы стандартные встроенные исключения Python). Последняя часть **except** в этом примере улавливает все другие исключения, которые не были пойманы выше. Например, если хотя бы в одном из файлов находится нечисловое значение, функция int() возбудит исключение ValueError. Его-то и сможет отловить последняя часть **except.** Разумеется, выполнение части **try** в случае возникновения ошибки уже не продолжается после выполнения одной из частей **except.**

5. Отладка программы

Процесс написания программы состоит из двух этапов: кодирование и отладки. В процессе отладки программы возможны различные ошибки.

Все ошибки можно условно разделить на следующие три категории (рис.2):

1) ошибки, выявляемые препроцессором

В интерпретатор Python встроена специальная программа - препроцессор. У препроцессора несколько функций, в частности, он переводит текст программы в специальный байт-код, понятный для интерпретатора. В процессе перевода текста в байт-код препроцессор вынужден анализировать синтаксис вашей программы, для

чего используется синтаксический анализатор, проверяющий ваш текст с целью понять, похож ли он на текст программы на Python по ряду формальных признаков. Если препроцессор не может понять смысл тех или иных символов в вашем тексте, он чаще всего указывает вам на ошибку типа (SyntaxError).

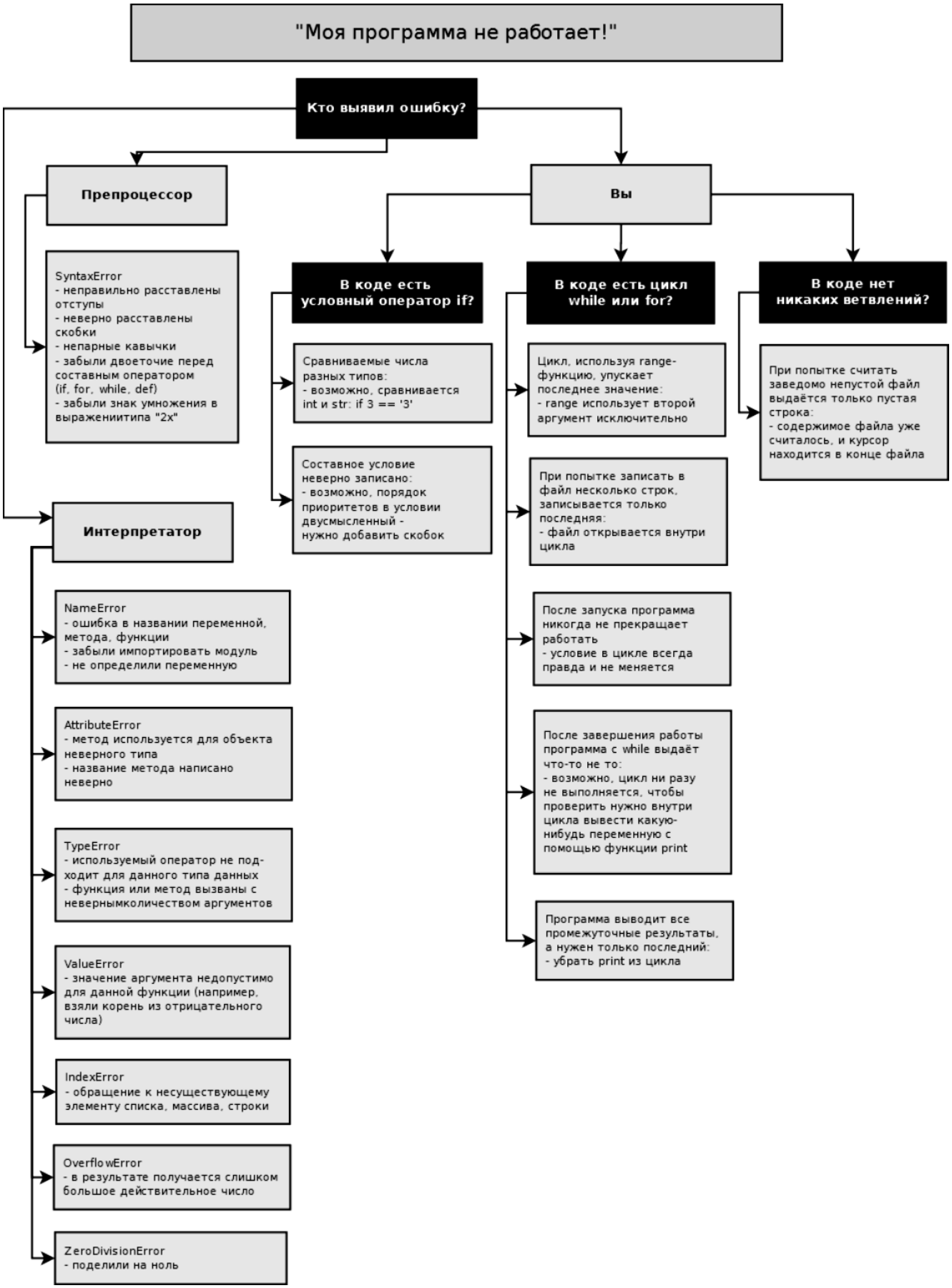

Рисунок 2 - Возможные источники ошибок и алгоритм их нахождения

2) ошибки, выявляемые интерпретатором

Интерпретатор выявит ошибки во время исполнения программы и напишет, что это за ошибка и в какой она строке кода.

3) ошибки, выявляемые разработчиком

Их ещё можно назвать логическими. Это такие ошибки, когда ваша программа работает, но выдаёт что-то не то. Это наиболее сложный тип ошибок, потому что их нужно не только устранять, но и выявлять самостоятельно.