PYTHON

Лекция 2. Структура программы. Управлящие конструкции

СТРУКТУРА ПРОГРАММЫ

- Формальная спецификация: https://docs.python.org/3/reference/index.html
- Подход отличается от многих языков
- Основные отлиичия: роль деления нп строки и роль отступов
- Ширина строки: холиварная тема
- (Строка здесь - элемент файла, не строковая константа)

ФИЗИЧЕСКАЯ СТРОКА

- Последовательность символов, завершающаяся переводом строки
- Три с половиной способа завершить физическую строку
  - LF, CR + LF, CR - можно на любой платформе
  - Конец файла - для последней строки

ЛОГИЧЕСКАЯ СТРОКА

- Состоит из 1+ физической строки с явным или неявным соединением (explicit/implicit joining)
- Явное соединение - через \ в конце физической строки

```
1 if 1900 < year < 2100 and 1 <= month <= 12 \
2    and 1 <= day <= 31 and 0 <= hour < 24 \
3    and 0 <= minute < 60 and 0 <= second < 60:
4         return 1
```

ЛОГИЧЕСКАЯ СТРОКА

- Неявное соединение - до закрываюей скобки (круглой, квадратной, фигурной)

```
1 if (1900 < year < 2100 and 1 <= month <= 12
2    and 1 <= day <= 31 and 0 <= hour < 24
3    and 0 <= minute < 60 and 0 <= second < 60):
4         return 1
```

ЕЩЕ ПРИМЕР

```
1 month_names = ['Januari', 'Februari', 'Maart',
2                'April',   'Mei',      'Juni',
3                'Juli',    'Augustus', 'September',
4                'Oktober', 'November', 'December']
```

ПУСТЫЕ ЛОГИЧЕСКИЕ СТРОКИ

- Легальный частный случай
- Есть нюанс с итерактивной средой
- Абсолютно пустая логическая строка воспринимается как конец ввода
- Важно понимать при копировании кусков кода

СМЕЩЕНИЯ

- Смещение - количество пробельных символов в начале логической строки
- Вместо пробелов могут быть табуляции - но нельзя смешивать
- А лучше - вообще без табуляции
- (Tab в IDE часто порождает пробелы)

СМЕЩЕНИЯ

- Интерпретатор хранит стек смещений
- В начале на стеке 0
- Логические строки обрабатыываются по порядку
- Для каждой считается логическое смещение

СМЕЩЕНИЯ

- Если оно равно значению на стеке - все хорошо, стек не трогаем
- Если больше - кладем новое значение на стек
- Если меньше - ищем на стеке самое верхнее вхождение такого значения ("перебираем свепху вниз")
- Если дошли до меньшего - сигнализируем ошибку
- Если дошли до искомого - сбрасываекм со стека все, что выше

СМЕЩЕНИЯ И СИНТАКСИЧЕСКАЯ ИЕРАРХИЯ

- Момент добавления элемента на стек - начало блока
- Момент снятия - завершение как минимум одного блока
- Завести блок "просто так" нельзя
- Да и смысла большого нет - нет локальных переменных уровня блока
- Если очень хочется - можно через `if True:`

ПУСТЫЕ БЛОКИ

- Пустых блоков не задать
- Но иногда они нужны (например, как временные заглушки)
- Есть ключевое слово pass

```
1 def f():
2     pass
```

ПУСТЫЕ СТРОКИ И КОММЕНТАРИИ

ПРИМЕР

```
 1 def perm(l):
 2         # Compute the list of all permutations of l
 3     # corner case
 4     if len(l) <= 1:
 5                   return [l]
 6     r = []
 7     for i in range(len(l)):
 8              s = l[:i] + l[i+1:]
 9              p = perm(s)
10              for x in p:
11               r.append(l[i:i+1] + x)
12     return r
```

ЕЩЕ ПРИМЕР

```
 def perm(l):   # error: first line indented
for i in range(len(l)):   # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])
        # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r
            # error: inconsistent dedent
```

УСЛОВНОЕ ИСПОЛНЕНИЕ

- Базовое ключевое слово - if
- На первой логической строке - if и условие
- Потом блок для True
- Если нужен блок для False - пишем ключевое слово else между блоками

ПРИМЕР

```
 1 if a % b == 0:
 2     print("b divides a")
 3
 4 if s = 'vasya':
 5     print("hello, dear", s)
 6 else:
 7     print("hello", s)
 8
 9 if s = 'vasya':
10     prefix = "hello, dear"
11 else:
12     prefix = "hello"
```

УСЛОВНОЕ ИСПОЛНЕНИЕ

- Хотим написать функцию, которая по числу возвращает существительное с согласованным окончанием
- Существительное - константа (например, "попугай")
- Число - параметр функции
- Для 0 возвращаем "попугаев", для 12 - "попугаев", для 2 - "попугая"
- 22 - "попугаев", 102 - "попугая"

УСЛОВНОЕ ИСПОЛНЕНИЕ

- Надо учесть последнюю цифру
- Отдельно обработать ситуацию - "вторая цифра - 1"
- Отдельно - "вторая цифра - 0"
- Писать вложенные if - не очень красиво
- Особенно если вложенность - в одной ветке (что часто бывает - когда пошагово применяем разные критерии)

ELIF, MATCH

- Классическое решение - elif
- Сливаем else и if вместе и остаемся на том же уровне смещения
- Альтернатива с 3.10 - match
- Но в нем свои нюансы

ПРИМЕР

```
1 if n % 10 == 1 && n % 100 != 11:
2     print("попугай")
3 elif n % 10 in [2, 3] && n % 100 not in [12, 13]:
4     print("попугая")
5 else:
6     print("попугаев")
```

MATCH

- match, потом выражение, двоеточие
- Вложенные case-блоки
- По блоку на вариант значения
- Ключевого слова default нет
- Ветка по умолчанию выражается через case

ПРИМЕР

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

ЧУТЬ ПО-ДРУГОМУ

```
 1 def http_error(status):
 2     match status:
 3         case 400:
 4             return "Bad request"
 5         case 404:
 6             return "Not found"
 7         case 418:
 8             return "I'm a teapot"
 9         case v:
10             return "Something's wrong with " + \
11                    "the internet: %d" % (v, )
```

MATCH

- Ищем первую сработавшую ветку
- Если нашли - следующие не исполняем, break не нужен
- Если ни одно не сработала - это не ошибка
- Примерно как несработавший if без else

GUARD

- К любой case-ветке можно добавить if с выражением
- В выражении можно использовать переменную из case
- Переменная может быть не одна
- Если guard не сработал - продолжаем перебирать case-ветки

ПРИМЕР

```
1 match (n % 10, n // 10 % 10):
2   case (a, b) if a == 1 and b != 1:
3     print("попугай")
4   case (a, b) if a in [2, 3] and b != 1:
5     print("попугая")
6   case _:
7     print("попугаев")
```

ЦИКЛЫ: ОБЗОР

- Два вида циклов - while и for (for препочтительнее)
- Нет в явном виде циклов с пост-условием
- Альтернативные формы итерирования: генераторы, списочные выражения
- else с цмклом

WHILE

- В первом приближении - синтаксический эквивалент if
- Только ключевое слово другое
- И семантика повторяющейся проверки условия
- for - более предпочтительная форма цикла
- while - более мощная

FOR

- Итерирование по последовательностям - в обобщенном смысле этого слова
- Примеры последовательностей: списки, строки, кортежи, словари, множества
- Примеры последовательностей: range, генераторы, итераторы
- Перебирает элемнты последовательности
- На каждой итерации присваивает значение переменной цикла

ПРИМЕР

```
 1 # цикл здорового человека
 2 for c in 'hello':
 3     print("code:", ord(c))
 4
 5 # цикл курильщика
 6 i = 0
 7 s = 'hello'
 8 while i < len(s):
 9     c = s[i]
10     print("code:", ord(c))
11     i += 1
```

МИНУСЫ WHILE-ВАРИАНТА

- Текстуально длиннее
- Располагает к ошибкам
- Идея теряется за деталями
- Изобретаем велосипед

RANGE

- range - функция-конструктор
- Создает объект типа "диапазон целых"
- Число параметров - от 1 до 3 включительно
- range(n) - полуоткрытый диапазон [0, n)
- Пустой, если n <= 0

RANGE

- range(lwb, upb) - полуоткрытый диапазон [lwb, upb)
- Пустой, если upb <= lwb
- range(lwb, upb, step) - берем числа с шагом step
- step не имеет права быть 0 - даже если lwb == upb

RANGE

- Если step > 0, то попадают числа lwb, lwb + step, lwb + i * step, пока меньше, чеи upb
- Если step < 0, то попадают числа lwb, lwb - step, lwb - i * step, пока больше, чеи upb
- Параметры - только int
- (Задумаемся о том, как это обеспечивается)

ПРИМЕР

```
 1 for i in range(10): # 0, 1, 2, ...., 9
 2     print(i, i * i)
 3
 4 for i in range(5, 10): # 5, 6, 7, 8, 9
 5     print(i, i * i)
 6
 7 for i in range(1, 10, 2): # 1, 3, 5, 7, 9
 8     print(i, i * i)
 9
10 for i in range(10, -1, -2):
11     # 10, 8, 6, 4, 2, 0 - не проблемы с -1
12     print(i, i * i)
```

RANGE

- Все в Python - объект
- И range - тоже
- Его можно присваивать переменной и передавать параметром в функцию
- И даже использовать как ключ в словаре
- И у него реализовано умное сравнение

WHILE VS FOR

- Когда полезен while
  - Потенциально бесконечный цикл типа "запрос" - "ответ"
  - Нелинейные прыжки по коллекции: бинарный поиск и т.п.
  - Алгоритмические трюки: например, двойной индекс
  - Нелокальный фокус

BREAK

- Выход из цикла заранее
- Обычно под if
- Типичное применение: чего-то ждали и вот дождались, уходим
- Действует на самый внутренний цикл

ПРИМЕР

```
1 def print_primes(n):
2     for i in range(2, n):
3         for j in range(2, i):
4             if i % j == 0:
5                 break
6             if j * j > i:
7                 print(i)
8                 break
```

ELSE

- Типичный шаблон: цикл ради поиска
- Можем что-то найти или не найти
- Хотим сделать какое-то действие на случай, когда не нашли
- (Сложнее, чем вернуть значение по умолчанию)

ПРИМЕР

```
 1 def example(n):
 2     while True:
 3         s = input()
 4         for c in s:
 5             if f(c):
 6                 print(s, "OK")
 7                 break
 8         else:
 9             break
10             # выйдем из внешнего цикла,
11             # если для всех символов s f вернуло False
```

CONTINUE

- Если сложное тело цикла
- Но для какого-то случая оно неактуально
- Тоже только на внутренний цикл
- Для внешнего нужно break из внутреннего с последующим условным continue
- Или выносить внутренний цикл в функцию