Лекция 4

Условия и циклы

Инструкции и операторы

- Символы операций употребляемые в выражениях, и сами эти операции называют *операторами* (operator)
- Конструкции управляющие порядком выполнения программы называют *инструкциями* (statement)
- Основные управляющие инструкции являются составными конструкциями языка, т.е. состоят из строки заголовка и блока
  - *заголовок* это строка, заканчивающаяся двоеточием
  - *блок* это последовательность строк текста программы, отступ которых на единицу больше, чем у строки-заголовка
- Отступы составлены из пробелов или горизонтальных табуляций. Не стоит (в версии 3 нельзя) смешивать пробелы и табуляции в отступах.
- Рекомендуемый единичный отступ 4 пробела (PEP 8)
- Отступ создается нажатием на клавишу табуляции. Текстовый редактор заполняет (или не заполняет) ее пробелами.

Пример синтаксиса составной конструкции языка

```
if apples > pears:
  eat_apples(apples - pears)
  print('We ate some apples')
else:
  eat_pears(pears - apples)
  print('We ate some pears')
get_more_fruits()
```

- В примере показан условная инструкция if с альтернативным блоком else
- Обе строки вводящие идущий за ними блок заканчиваются двоеточием
- Вызов функции get_more_fruits() находится на том же уровне, что и заголовки инструкции if и не входит в ее состав

Компактная запись

- В строке можно разместить несколько инструкций используя точку с запятой в качестве *разделителя*
- Если блок составной инструкции можно уместить в одну строку его можно вписать в строку заголовка сразу после двоеточия
- Примеры компактной записи

```
a = 1; b = 2; c = a + b; print(c)

if c < 5: a = 5 ; print(a)
if c > 5: a = 9 ; print(a)
```

- Однако два двоеточия в строке недопустимы

```
if c < 5: print('less') else: print('greater') # Ошибка

if c < 5: print('less') # Запись в две строки
else: print('greater')  # является допустимой
```

Инструкция if. Проверка условия и ветвление.

- Синтаксис:

```
if условие_1:
   блок_инструкций_1
elif условие_2:
   блок_инструкций_2
else:
   блок_инструкций_3
```

  Конструкции elif и else необязательны

- Пример:

```
if t < 0:
   t = 0
elif t > 100:
   t = 100
else:
   t = fix_drift(t)
```

Несколько блоков elif

```
s = 'three'
if s == 'zero' or s == 'null' or s == 'nil':
  print(0)
elif s == 'one':
  print(1)
elif s == 'two':
  print(2)
elif s == 'three':
  print(3)
elif s == 'four':
  print(4)
elif s == 'five':
  print(5)
else:
  print(-1)
```

- Проверки условий производятся последовательно до первого совпадения. В этом примере проверки n == 'four' и n == 'five' производиться не будут
- В Питоне нет инструкции switch / case

Альтернатива нескольким блокам elif

```
d = {
  'zero':  0,
  'null':  0,
  'nil':   0,
  'one':   1,
  'two':   2,
  'three': 3,
  'four':  4,
  'five':  5
}
print(d.get('three', -1))  # =>  3
print(d.get('twenty', -1)) # => -1
print(d.get('fifty')) # => None
```

- Второй параметр функции get() это значение возвращаемое для ненайденного ключа (значение по умолчанию)
- Код с использованием dict при большом числе альтернатив выбора работает быстрее

Условия

- Элемент *условие* в инструкции if это выражение, возвращающее объект типа bool или автоматически приводимый к типу bool
- Тип bool имеет два значения, которые можно задать литералами True и False
- Числа равные нулю приводятся к значению False
- Коллекции
  - пустые коллекции приводятся к значению False
  - непустые коллекции приводятся к значению True
- Строки и байтовые массивы это разновидность коллекций
  - пустая строка (массив) приводится к значению False,
  - строка (массив) содержащая хотя бы один символ (байт) приводится к значению True
- Явное приведение к типу bool производится вызовом встроенной функции bool()

Примеры использования and и or

- Операторы *and* и *or* выполняются последовательно, до тех пор, пока результат не будет очевиден. Ненужные действия произведены не будут. *Это справедливо и для языка C*.

```
def less2(n):
  if n < 2: return True
  else: return False

a = less2(1) and less2(2) and less2(3) and less2(4) # => False
b = less2(2)  or less2(1)  or less2(0)  or less2(4) # => True
```

- При вычислении выражений для переменных a и b будут выполнены только первые два вызова функции less2()

- Примеры использования and и or вместо инструкции if

```
проверить_корректность_объекта(obj) and использовать_объект(obj)

была_ли_ошибка(obj) or обработать_ошибку(obj)
```

Особенности операторов and и or

- Результат оператора and или or это не величина типа bool, а последний объект, участвовавший в операции и ставший причиной принятия решения

```
a = [] or { 1, 2, 3 } or 'LastToken' # => { 1, 2, 3 }
b = { 1, 2, 3 } or [] or 'LastToken' # => { 1, 2, 3 }
c = [ 1, 2 ] and { 1, 2, 3 } and 'LastToken' # => 'LastToken'
d = [ 1, 2 ] and [] and 'LastToken' # => []
```

- Приоритет оператора and выше, чем оператора or

```
 1 or 2  and  0 or 3  # => 1
(1 or 2) and (0 or 3) # => 3
```

- Результат оператора not величина типа bool

```
e = [ 1, 2 ] and not { 1, 2, 3 } # => False
```

Трехместный оператор if/else

- Оператор if/else в отличие от инструкции if используется в выражениях и сам является выражением
- По своему действию оператор if/else близок к операторам and и or, но более нагляден
- Пример:

```
x = -3
sign = 'plus' if x >= 0 else 'minus' # => 'minus'

print('The result is ' + ('plus' if x >= 0 else 'minus') +
' ' + str(x if x >= 0 else -x)) # => 'The result is minus 3'
```

- *Нет двоеточий* после условия if и ключевого слова else
- Ключевое слово else является обязательным
- Трехместный оператор if/else имеет очень низкий приоритет, в выражениях его всегда следует помещать в скобки

Инструкция присваивания

- В отличие от языка C в Питоне *присваивание это инструкция*
- Пример *на языке C,* присваивание это *оператор* и может быть использован в выражении

```
a = b = (c = 4) + (d = 3 + 2);
```

- В Питоне инструкция присваивания имеет левую и правую часть. Символ присваивания их разделяет и может быть только один.
- В *левой* части инструкции присваивания может находиться:
  - переменная: x
  - атрибут объекта: x.attr
  - выборка элемента: a[n]
  - выборка фрагмента: a[n:m:k]
- В *правой* части инструкции присваивания находится *выражение*

Расширенная инструкция присваивания

- Расширенная инструкция присваивания или *присваивание последовательностей* (sequence assignment) позволяет в левой и правой части присваивания использовать последовательности
- Присваивание последовательностей происходит поэлементно
- Последовательность в левой части может быть оформлена как кортеж или как список. Если не возникает двусмысленности литерал кортежа можно записать без скобок.

```
(a, b, c, d, e) = (1, 2, 3, 4, 5)
 a, b, c, d, e  = (1, 2, 3, 4, 5)
[a, b, c, d, e] = (1, 2, 3, 4, 5)
```

- Результат всех трех конструкций эквивалентен следующей записи

```
a = 1 ; b = 2 ; c = 3 ; d = 4 ; e = 5
```

Расширенная инструкция присваивания

- Справа от символа присваивания может находиться любая коллекция

```
a, b, c, d, e = "abcde"
a, b, c, d, e = { 1, 2, 3, 4, 5 }
a, b, c, d, e = { 'a':1, 'b':2, 'c':3, 'd':4, 'e':5 }
```

- Если число элементов в правой части присваивания заранее неизвестно используется конструкция **переменная*
- Конструкция **переменная* поглощает неиспользованные элементы сохраняя их *в списке*:

```
a, *x, d, e = 1, 2, 3, 4, 5 # x => [ 2, 3 ], x это list
```

- Конструкция **переменная* может быть только одна
- В качестве элементов последовательности в *левой* части расширенной инструкции присваивания допустимы те же конструкции, что были рассмотрены ранее

Структурное программирование

- Теорема Бема-Якопини (1966)
  Любой исполняемый алгоритм может быть реализован
  с помощью трех управляющих структур
  - *последовательность* (sequence)
  - *ветвление* (selection)
  - *цикл* (cycle)
- Смысл теоремы в том, что после введения понятия цикла более общая управляющая структура *переход* стала ненужной
- В ряде языков программирования переход реализован инструкцией *goto*
- В Питоне инструкции goto нет
  - однако есть попытки ее ввести (см. репозиторий PyPi)
- Введение в язык *исключений* закрывает еще одну тему,
  где использование goto может быть полезным

Циклы ([1], глава 13)

- Цикл это многократное исполнение блока инструкций
- Два вида циклов и организующие их инструкции
  - Инструкция *while* - цикл по условию
  - Инструкция *for* - цикл по элементам коллекции
- Вспомогательные инструкции
  - Инструкция *break* прерывает цикл
  - Инструкция *continue* прерывает исполнение блока и начинает новый проход по циклу с первой инструкции блока
  - Инструкция *pass* не делает ничего, но из нее можно сделать пустой блок инструкций
- Циклы в Питоне могут иметь блок *else*

Инструкция while - цикл по условию

- Синтаксис:

```
while условие:
   блок_инструкций_1
else:
   блок_инструкций_2
```

  Конструкция else необязательна

- Пример:

```
n = 1
while n < 10:
   print('Loop:', n)
   n = n * 2
   if n == 8:
       break;
else:
   print('Else:', n)
```

  else выполняется если *цикл не был прерван* инструкцией break

Инструкции break и continue

- *break* "аварийно" завершает весь цикл
- *continue* "аварийно" завершает один проход по циклу и начинает следующий проход с начала, включая проверку условия (для while) или выборку нового элемента (для for)
- Инструкции break и continue почти всегда находятся внутри блока инструкции if

Смысл инструкции else в циклах

Цикл может завершиться успешно или быть прерван. В обоих случаях есть место, где можно записать группу инструкций, которая будет исполняться после окончания цикла одним из двух возможных способов.

- Для прерванного цикла - после положительной проверки на "аварию", непосредственно перед инструкцией break,
- Для нормального завершения - в блоке else

Инструкция pass

- Инструкция *pass* не делает ничего, но это инструкция
- Обычно инструкция pass используется как заглушка на месте кода, который будет написан позднее

```
def do_nothing():
  pass

a = 3
if a < 0:
  pass
else:
  print(a)
```

- Инструкция pass может быть использована для организации пустого цикла, имеющего побочный эффект

```
while eat_apple(fruits):
  pass
```

Объект ... (Ellipsis)

- Объект Ellipsis имеет свой собственный класс ellipsis
- Ellipsis используется для выборки фрагментов (slice) из многомерных массивов, реализованных посредством многократно вложенных списков
- В Питоне версии 3 объект Ellipsis можно использовать в конструкциях требующих pass или None

```
while eat_apple(fruits):
  ...

a = ...
if a == ...:
  print('a is really ...')
```

- В отличие от инструкции pass, Ellipsis это объект
- В отличие от None, Ellipsis приводится к логическому значению True

Инструкция for - цикл по коллекции

- *Коллекция* это объект, обладающий свойством перечислимости (enumerable). Коллекция *может не быть* последовательностью.
- Синтаксис инструкции перебора элементов коллекции:

```
for переменная in коллекция:
   блок_инструкций_1
else:
   блок_инструкций_2
```

Конструкция else необязательна

```
for e in (1, 2, 'three', 4, 5):
   print('Loop:', e)
   if e == 4:
       break;
else:
   print('Else:', e)
```

else выполняется если цикл не был прерван инструкцией break

Исполнение инструкции for

- Инструкция for перебирает элементы коллекции и исполняет блок столько раз, сколько элементов есть в коллекции, при условии, что исполнение не было прервано инструкцией break
- В начале исполнения блока перед исполнением первой его инструкцией происходит присваивание:

```
переменная_цикла = очередной_элемент_коллекции
```

- Исполнение цикла можно представить следующим образом:

```
for переменная_цикла in коллекция:
   переменная_цикла = выбрать_следующий_элемент()
   блок_инструкций

for e in (1, 2, 'three', 4, 5):
   # Здесь происходит присваивание переменной e
   print('Loop:', e)
   if e == 4:
       break;
```

Эмуляция цикла с целой переменной

- Для имитации числового ряда используется встроенная функция range

```
for i in range(10, 15):
  print(i)
```

Этот цикл напечатает числа 10, 11, 12, 13, 14
Второй параметр range - число следующее после последнего

- Необязательный третий параметр задает шаг последовательности

```
for i in range(10, 15, 2): print(i) # => 10, 12, 14
```

- Единственный параметр задает число следующее после последнего, при этом первым числом будет ноль

```
for i in range(3): print(i) # => 0, 1, 2
```

Использование for со словарями

- С точки зрения цикла for словарь это совокупность *ключей*
- Зная ключ можно получить значение элемента

```
fruits = { 'apples': 4, 'pears': 6, 'plums': 8 }
for key in fruits:
  print(key, '=', fruits[key])
```

```
Результат:
pears = 6
apples = 4
plums = 8
```

- Возможен перебор всех значений элементов словаря с использованием функции-атрибута values()

```
for value in fruits.values():
  print(value)
```

```
Результат: 6 4 8
```

Использование for со словарями

- Функция-атрибут items() позволяет перебрать элементы словаря в виде пар ключ-значение

```
for element in fruits.items():
  print(element)
```

- Тип выбираемых элементов - кортеж (tuple), *ключ* имеет индекс 0, *значение* индекс 1

```
for element in fruits.items():
  print('key =', element[0], 'value =', element[1])
```

- Присваивание кортежа в цикле for

```
for (key, value) in fruits.items():
  print('key =', key, 'value =', value)
```

- В инструкции for скобки кортежа необязательны

```
for key, value in fruits.items():
  print('key =', key, 'value =', value)
```

Соединение последовательностей

- Функция zip получает в качестве параметров последовательности и возвращает последовательность *кортежей* с элементами исходных последовательностей, имеющими равные индексы
- Функция zip может быть использована для создания словаря, из раздельно заданных последовательностей ключей и значений

```
names = 'apples', 'pears', 'plums'
values = 4, 6, 8
d = dict(zip(names, values)) # => {'plums': 8, 'apples': 4, 'pears': 6}
```

- Функция zip может соединить любое количество последовательностей

```
z = zip(names, values, 'abc', reversed(values))
z # => ('apples', 4, 'a', 8), ('pears', 6, 'b', 6), ('plums', 8, 'c', 4)
```

- Длина результата равна длине самой короткой из исходных последовательностей

Генераторы коллекций

- *Генератор коллекции* (comprehension) это компактный способ записи цикла, в результате исполнения которого создается последовательность
- Конструкция цикла находится внутри литерала последовательности и может быть использована с литералами кортежей, списков, словарей и множеств
- Синтаксис генератора коллекции:

```
[ выражение_1 for переменная_цикла in коллекция if условие else выражение_2 ]
```

- В зависимости от типа создаваемой коллекции в генераторе могут быть использованы круглые, квадратные или фигурные скобки
- Фрагмент программы эквивалентный генератору коллекции:

```
результат = []
for переменная_цикла in коллекция:
 if условие:                        # конструкция if необязательна
   результат.append(выражение_1)
 else:                              # внутри конструкции if
   результат.append(выражение_2)    # конструкция else необязательна
```

- Генератор коллекции это объект, ведущий себя как коллекция. То же касается значения, возвращаемого функцией zip

Использование генераторов

- Генератор коллекции это *выражение,* то есть генератор может быть операндом входящим в другое выражение
- Генератор коллекций позволяет создавать неизменяемые коллекции
- Пример: список и кортеж

```
a = [ -n if n < 0 else n + 10 for n in (-2, -1, 0, 1, 2) ]
print(a) # => [ 2, 1, 10, 11, 12 ]

d = { 'name': 'abcd', 'f1': 14, 5: 27.12, 'pair': [ 1, 2 ] }
c = ( print('key =', key, 'value =', value) or
  str(key) + '/' +  str(value)
  for key, value in d.items() )
print(c) # => ('f1/14', 'pair/[1, 2]', 'name/abcd', '5/27.12')
```

- Пример: словарь и множество

```
e = { 'char ' + chr(65 + i): i + 10 for i in range(4) }
print(e) # => {'char C': 12, 'char B': 11, 'char A': 10, 'char D': 13}

s = { i + 20 for i in range(4) } # => { 20, 21, 22, 23 }
```

Итераторы

- Итератор это объект, посредством которого можно последовательно получить доступ ко всем элементам коллекции
- Объект-коллекция сам для себя является итератором или, другими словами, содержит собственный итератор внутри себя
- Итератор в явном виде создается когда необходимо производить несколько независимых переборов элементов коллекции одновременно

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
i1 = iter(a)
i2 = iter(a)
while True:
  e1 = next(i1, None)
  if e1 == None:
    break
  if e1 % 2:
    e2 = next(i2)
  print(e1, e2)
```

Итераторы

- Итератор может быть передан в оператор for

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
i1 = iter(a)
i2 = iter(a)
for e1 in i1:
  if e1 % 2:
    e2 = next(i2)
  print(e1, e2)
```

- Одновременная выборка по коллекции и ее итератору

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
i2 = iter(a)
for e1 in a:
  if e1 % 2:
    e2 = next(i2)
  print(e1, e2)
```

Неявные циклы

Итератор является решением проблемы в случае неявно заданного цикла, т.е. цикл реализован для другой цели и в этом цикле требуется получение элементов коллекции.

Примеры:

- Цикл Internet сервера по обслуживанию сетевых соединений
- Цикл программы с графическим пользовательским интерфейсом для восприятия действий клавиатуры и мыши
- Цикл программы сбора данных для реакции на сообщения от измерительных датчиков

Неявные циклы могут находиться в библиотечных функциях. Программист может лишь реагировать на специальные сигналы, генерируемые внутри цикла.

Эволюция функции next

```
a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
i1 = iter(a)

e1 = i1.next() # Python 2.x, ранний вариант
e1 = i1.__next__() # Python 2.x, более поздний вариант, также 3.x
e1 = next(i1, default) # Python 3.x, в Python 2.x также работает
```

- Еще один пример: функция len() и метод __len__()
- Общая тенденция: переход от методов объекта (функций-атрибутов) к встроенным функциям
- Чем больше символов подчеркивания в начале или конце имени, тем более "внутренним" это имя является. Если есть выбор, используйте менее внутреннее имя.
- Имена с двумя подчеркиваниями и в начале и в конце это собственные имена интерпретатора. Такие имена следует создавать только при реализации собственных классов как интерфейс к интерпретатору.