Питон
Курс лекций
Лекция шестая

© Зайцев Вадим Валерьевич, 2002–2010,
spider.vz@gmail.com

Несмотря на то, что питон — язык нетипизированный, мы и в этой лекции рассмотрим ещё один тип данных и операторы, им порождённые.

На практике иногда оказывается, что типы, кажущиеся на первый взгляд менее важными, да и по определению скромнее уже рассмотренных, во много крат мощнее и нужнее. Таков, например, так называемый логический тип, называемый иногда булевым в честь ирландского математика Джорджа Буля.

3.10 Логический тип

Логический тип, как можно догадаться, состоит всего из двух возможных значений: истины и лжи (англоязычные источники пользуются терминами *true* и *false* соответственно). В питоне нет самостоятельного булева типа даже на том уровне, где можно признать существование целого и вещественного типов данных (ведь, вообще говоря, нет никаких типов, так, теория одна). В качестве булева типа возможно использование любого другого по следующим правилам:

- Операции отношения, такие, как `>`,`==` или `!=`, возвращают `1` (значение целого типа), если отношение выполняется и `0` в противном случае.
- При использовании значения какого-либо типа в качестве булева только всевозможные нули и пустые списки (то есть `0, 0.0, 0L, 0j, (), '', , '""', , [], {}`, а также особый пустой объект `None`, с которым мы ознакомимся позже) считаются ложью, все прочие — истиной.
- Выражение `A or B` (A или B) возвращает B, если A ложно и A в противном случае.
- Выражение `A and B` (A и B) возвращает B, если A истинно и A в противном случае.

- Выражение `not A` (не A) возвращает 1, если A ложно и 0 в противном случае.

Изменить это (если кому вдруг захочется) нельзя, а создать новый тип с похожими свойствами можно только пользуясь объектно-ориентированным подходом, о чём мы расскажем позже.

Булев тип чаще всего используется при различного рода проверках, в операторах ветвления, которые мы сейчас рассмотри. Когда мы говорили об операторе присваивания, было упомянуто существование операторов, управляющих последовательностью их выполнения. Теперь же мы обсудим их подробно.

В языке питон существует три вида таких операторов: ветвление, повтор и перебор. Оператор ветвления записывается так:

```
if <условие>: <оператор>
```

или так:

```
if <условие>:
␣<оператор>
```

После ключевого слова `if` записывается условие (для наглядного отделения обычно используют круглые скобки, но можно обходиться без них). Вообще, скобок можно ставить сколько угодно, питон не спутает число в скобках с кортежем из одного элемента, ведь такой кортеж должен в задании иметь ещё и запятую: `a=(0,)`.

После двоеточия указывается оператор, который будет выполнен в случае истинности условия. Если в случае истинности нужно выполнить не одну, а несколько строк кода, используется так называемый составной оператор, обозначаемый отступом:

```
if␣(A):
␣B=input()
␣print␣A+B
```

Отступом может служить как пробел, так и символ табуляции. Составной оператор (а с ним и оператор ветвления) кончается перед следующей строкой без отступа.

Для сравнения величин многих типов (чисел, строк, ...) используются привычные математические символы: `<` для *меньше*, `>` для *больше*, `>=` для *больше или равно*, `<=` для *меньше или равно*, `==` для *равно*, `<>` или `!=` для *не равно*. Их можно группировать по всем правилам арифметики:

```
if 0<x<10 and -10<=y<=100:
 print y%x
```

Есть ещё две инфиксных (то есть записывающихся между операндами) операции сравнения: `is` и `in`. Первая используется в основном для сравнения объектов на эквивалентность, для более простых же типов данных она аналогична `==`. Вторая проверяет элемент на принадлежность последовательности (кортежу, списку или строке). Есть краткая запись для `not (e in L)`:

```
e not in L
```

Оператор ветвления можно продлить, добавив секцию, срабатывающую при **ложном** условии. Повторно указывать условие не нужно:

```
if (<условие>):
␣<операторы>
else:
␣<операторы>
```

Развивая заложенную в этом маленьком усовершенствовании большую идею, можно придти к так называемому множественному ветвлению, когда в случае неудачи одного условия проверяется другое, при его неудаче — третье, четвёртое, и так далее. В питоне это записывается следующим образом:

```
if (<условие>):
␣<операторы>
elif (<условие>):
␣<операторы>
elif (<условие>):
␣<операторы>
else:
␣<операторы>
```

Логические операции, которым мы дали определение в начале лекции, помогают существенно сократить или даже полностью избежать появления одинаковых блоков программы или одинаковых условий:

```
if (A):
 print "!"
elif (B):
 print "!"
else
 print "?"
```

$\Longrightarrow$

```
if (A or B):
 print "!"
else
 print "?"
```

Совет, данный нами при описании приоритетов различных операций, остаётся в силе и здесь: не жалейте скобок для того, чтобы сделать выражение более удобным и читабельным для вас — питон всё поймёт и всё простит, но простите ли вы себя сами через месяц, пытаясь разобраться в мудрёных условиях?

3.11 Комментарии

Комментариями называют части программы, не интересующие интерпретатор. В питоне есть два варианта комментариев: однострочные естественные и многострочные синтаксические. Комментарии первого типа начинаются символом # и завершаются переходом на новую строку.

```
i = 1 #этого питон уже не видит
```

Комментарии второго типа представляют собой строку, записанную без всякого присваивания. В случае прямой работы с интерпретатором в

диалоговом режиме эта строка будет выдана на экран, но при выполнении программы из файла она не попадёт никуда:

```
j = 1+i
"
Комментарий, поясняющий,
что в этом месте программы
переменная j
получила инкрементированное значение
переменной i
"
```

В новых версиях питона этот возникший чисто синтаксический механизм обмана интерпретатора получил более оправданное применение. Например, при определении функции комментарий записывается в специальную связанную с ней переменную `func_doc`.

4 Циклы и функции

4.1 Оператор перебора и оператор с предусловием

Оператор перебора позволяет применять одну и ту же последовательность операторов ко всем значениям последовательности. Записывается он так:

```
for x in (1,3,5,7,11,13,17,19):
␣<операторы>
```

при выполнении этого кода операторы будут выполнены столько раз, какова длина последовательности (в нашем случае это 8) и каждый раз `x` будет иметь значение очередного элемента последовательности: 1 на первом витке, 3 — на втором, 5 — на третьем, и т.д. Питон позволяет выполнять оператор перебора относительно нескольких переменных:

```
for x,y in ((1,2),(3,4),(5,6)):
␣<операторы>
```

При этом на каждом проходе пара `x` и `y` (точнее, кортеж, состоящий из этой пары) будет принимать значение соответствующей пары последовательности. Если структура последовательности не подходит, интерпретатор питона выдаст ошибку: Распаковка не-последовательности (*unpack non-sequence*).

Конечно, каждый раз указывать все значения — дело достаточно утомительное, поэтому в питоне есть встроеная функция `range()`, генерирующая список последовательных целых чисел в нужном интервале. С этой функцией мы мельком познакомились ещё в прошлой лекции. Эту чрезвычайно полезную функцию можно использовать тремя способами:

`range(n)`
создаст список целых чисел от 0 включительно до `n` невключительно.
`range(f,t)`
создаст список целых чисел от `f` включительно до `t` невключительно.
`range(f,t,s)`

создаст список чисел из интервала [f,t) вида *f, f+s, f+2*s, ...* — может быть полезно при использовании вещественных чисел.

При использовании чрезвычайно больших списков ради экономии памяти можно воспользоваться функцией `xrange()`, которая, работая абсолютно аналогичным образом, не вычисляет сразу значение каждого элемента итоговой последовательности, а создаёт определённый объект, элементы которого вычисляются только при непосредственном обращении к ним. Математик сказал бы, что `range()` реализует абстракцию актуальной бесконечности, тогда как `xrange()` — абстракцию потенциальной достижимости.

Если ваш список содержит несколько миллионов элементов, а одновременно нужны из них бывают только два или три, вы сможете заметить разницу в скорости выполнения программы при переходе с `range()` на `xrange()` невооружённым взглядом. Например, программа

```
from whrandom import choice
from time import clock
beg=clock()
A=range(30000000)
b=choice(A)
print clock()-beg
```

выполняется на компьютере AMD Duron 750MHz с 256Mb оперативной памяти и операционной системой Windows за 65-75 секунд, не считая пяти, а то и десяти минут выгрузки интерпретатора операционной системой, тогда как версия с `xrange()` выполняется за немногим более одной десятитысячной доли секунды.

Упражнение. Придумайте пример, когда время работы программы не может существенно измениться при переходе с `range()` на `xrange()`.

Но вернёмся к операторам циклов. Более высокоуровневую абстракцию повторяющихся операторов представляет собой цикл while — цикл с предусловием. При его использовании вместо прямого перечисления всех пробегаемых значений переменной цикла программист формулирует условие, которое остаётся истинным, если нужно выполнять итерацию и становится ложным в противном случае. Существуют языки программирования, специально ориентированные на такие условия (там они называются инвариантами). Все алгоритмы для программирования на подобных языках должны быть переформулированы с определение инварианта для каждой не единожды выполняемой строчки. Так далеко решаются заходить немногие, но циклы с условиями уже успели стать неотъемлимой частью всех алгоритмических языков.

Записывается цикл с предусловием так:

```
while <условие>
␣<операторы>
```

И, пока (а именно так, как вы знаете, переводится слово while) условие будет истинно, операторы будут выполняться ещё и ещё. Интерпретатор действует следующим образом: сначала проверяется условие и, если оно ложно, управление передаётся оператору, следующему за циклом while

(говорят: *происходит выход из цикла*). Если же условие истинно, выпл-дняются все операторы цикла (которые, как известно, находятся в отступе относительно самого оператора), после чего опять проверяется условие и в случае его истинности всё повторяется с начала, а в случае ложности происходит выход из цикла.

Очевидно, что цикл

```
while (1): ...
```

будет вечным (из него никогда не будет выхода), а цикл

```
while (0): ...
```

не будет выполнен ни разу. Цикл не может быть пустым, в случае необходимости используют ничего не делающий оператор pass:

```
while (1): pass # вечное бездействие
```

Для экстренного выхода из цикла также существуют особые методы. Для безусловного выхода используется всевдооператор break:

```
while (1):
␣i/=10
␣if␣(!i):␣break
```

Теперь становится обоснованным применение вечных циклов, не так ли?

Для условного выхода (ещё называемого продолжением вычислений) используют continue. Встретив этот псевдооператор, интерпретатор передаёт управление в точку, где происходит проверка условия цикла. Таким образом, при ложном условии continue вызывает выход из цикла, а при истинном — очередной виток вычислений.

Ясно, что средства break и continue применимы и к циклу перебора: первый прерывает цикл, а второй вызывает новый виток вычислений, если текущее значение переменной цикла не последнее, иначе также завершает перебор.

Оператор перебора и цикл с предусловием слабо эквивалентны, то есть для каждого конкретного условия будет достаточно легко перейти от одного типа цикла к другому, а общее преобразование куда сложнее. Из for в while можно построить автоматическое преобразование, которое будет неэффективным, а из while в for это вообще осуществить невозможно. Несмотря на столь очевидную связь, подобные мысли о взаимозаменяемости for и while концептуально категорически недопустимы. Эти циклы соответствуют абсолютно разным подходам к реализации вычислений: немедленные (for) и т.н. отложенные (while) вычисления. В качестве другого примера отложенных вычислений можно привести уже изученную нами функцию `xrange()`.