Питон
Курс лекций
Лекция четвёртая

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

3.4 Целые числа и операции над ними

Понятно, что при объяснении даже нетипизированного языка приходится уделять некоторое внимание типам данных, в нём используемых. Основной простейший тип данных — это, конечно, целые числа.

Целые числа в питоне бывают двух типов: обычные и длинные. Обычные занимают только 32 бита и могут иметь знак, т§о есть каждое целое число есть число от `-2147483648` до `2147483647` включительно. Задаются они следующим естественным образом:

```
a=65535
```

Конечно, существует возможность задания целых чисел и менее естественными способами, например, в восьмеричном или шестнадцатиричном виде. В первом случае число должно начинаться с нуля (и быть не более 017777777777), во втором — с `0x` (ограничено сверху 0x7fffffff). Недесятичные числа также могут иметь знак, но отрицательные числа могут быть также заданы явным представлением (например, -20 — это 037777777754 или же 0xffffffec). Получить строки с восьмеричным или шестнадцатиричным представлением числа можно функциями `oct` и `hex` соответственно.

Длинные целые числа имеют неограниченную длину (точнее, ограниченую объёмом всей оперативной памяти, а это очень и очень много). Отличаются длинные целые от обычных целых буквой `L` после последней цифры. Например:

```
b=99999999999999999999999999999999999999999999999999L
b=0xffffffffffffffffffffffffffffffffffffffffffffffffL
```

Все операции над целыми числами можно разделить на арифметические и логические. Арифметические — это с детства знакомые нам сложение (`+`), вычитание (`-`), деление (`/`, с округлением вниз), умножение (`*`) и возведение в степень (`**`), логические — это всевозможные побитовые операции НЕ (∼), И (`&`), ИЛИ (`|`), исключающее или (∧) и так далее. Для того, чтобы избежать путаницы между этими побитовыми операциями и логиче-

скими булевыми, последние были названы словами, а побитовым досталось по символу, их обозначающему.

```
c=2+3
c*=2
```

В последней сточке используется так называемое *присваивание умножением*, когда операция проводится напрямую с содержимым переменной `c`. Это эквивалентно:

```
c=c*2
```

Считается, что использование таких приёмов (которые существуют для всех операций: есть и присваивание делением, и возведением в степень, и исключающим или) экономит машинное время, позволяя более эффективно проводить вычисления, но очевидно, что это экономит как минимум несколько символов в исходном тексте программы.

3.5 Вещественные числа

Вещественные числа — это числа, имеющие наряду с целой ещё и дробную часть, которая приписывается справа от целой после точки. Они могут записываться тремя различными способами: в обычной, усечённой и экспоненциальной форме. Обычная форма подразумевает следующее:

```
d=-324.8451
e=2.7183
```

Усечённая форма позволяет записывать целые числа как вещественные (возможно, для того, чтобы к ним можно было применять вещественные функции). Индикатором «вещественности» числа в этом случае будет служить точка:

```
f=10.
```

При проверке на равенство такое число будет равно обычной, целой десятке, но при попытке разделить его на 100 мы получим не ноль, а одну десятую.

Аналогичным образом можно опускать и целую часть, записывая только точку и следующие за ней цифры.

Экспоненциальная форма позволяет дописывать в конце множитель. Дело в том, что вещественные числа хранятся в памяти ЭВМ не точно, а приближённо только несколько первых знаков после запятой (обычно 16), и для того, чтобы сделать такое представление не совсем уж плохим, используется множитель. Например, число 123456789123456789 будет иметь такой вещественный вид:

$$123456789123456789 \approx 1.2345678912345678 \cdot 10^{17}$$

Его можно будет записать в питоне так:

```
g=123456789123456789.
```

или:

```
g=1.23456789123456789e17
```

Понятно, что если число имеет немного значащих цифр, но большой порядок, его запись в экспоненциальной форме наиболее легка. Сравните эти два присваивания:

```
h=100000000000000000000000000000000000000000000000000.
h=1e50
```

Они абсолютно эквивалентны, в чём легко убедиться, посчитав нули.

Побитовых действий над вещественными числами не производится также ввиду сложности их машинного представления. Арифметические действия выполняются без округления, хотя погрешность может возникнуть, если, например, результат содержит много разреженых чисел. Например:

```
k=1.
l=1e-20
m=k+l
```

После этого `m`, как это ни печально, будет равно `k`, а вся добавочная часть в лице `l`, способная на отдельное существование, канет в небытие.

В добавление к обычным операциям, для работы над вещественными числами написан специальный модуль под названием `math`. Подключив его, можно будет считать всевозможные синусы, косинусы и тангенсы с арктангенсами. Ещё он позволяет правильно округлять числа и брать логарифмы. Помимо этого, модуль содержит две константы: e и π. Модуль подключается ключевым словом `import`, и далее при использовании любого оператора и любой константы оттуда нужно к имени добавлять имя модуля. Список всего содержимого модуля можно получить удобной функцией `dir`:

```
>>> import math
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2',
'ceil', 'cos', 'cosh', 'e', 'exp', 'fabs', 'floor', 'fmod',
'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow',
'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>> math.cos(math.pi)
-1.0
>>>
```

3.6 Комплексные числа

Питон имеет встроенную поддержку комплексных чисел. Их можно задавать двумя способами: прямым или же функцией `complex`. Делается это так:

```
n=2+1j
p=complex(2,1)
```

Эти две строчки эквивалентны. Обратите внимание на то, что в питоне мнимая единица обозначается маленькой латинской `j` и никак иначе. Причём эта буква вопринимается таким образом только если написана непосредственно после числа, поэтому даже в нашем случае нужно писать `1j`, а не `j`.

Обе части комплексного числа (как реальная, так и мнимая) считаются вещественными, даже если заданы в виде целого числа. Комплексное число без мнимой части не перестаёт быть комплексным числом.

Все приёмы работы с комплексным числом можно узнать той же функцией `dir`, дав ей комплексное число в качестве аргумента. Они включают: `real` для получения реальной части числа, `imag` соответственно для мнимой и `conjugate()` для получения комплексного числа, сопряжённого данному. Модуль комплексного числа (как и модуль любого другого) можно получить функцией `abs()`.

3.7 Связь между числами, связь между операциями

Если в одном и том же выражении встречаются несколько типов чисел, то результат имеет самый сильный тип из всех встречающихся в выражении. Самым сильным числовым типом считается комплексный, за ним идёт вещественный, затем длинный целый и только потом целый. Таким образом, присутствие дробной части считается более важным, чем точность большого числа. Стоит быть осторожным, если для вас это не так, и округлять все вещественные числа перед добавлением к длинным целым.

Если в одном и том же выражении встречаются несколько операций, то они, конечно, выполняются все, причём в определённой последовательности. Она жёстко определяется приоритетами операций и порядком их следования в выражении. Приоритеры таковы: самый высокий у возведения в степень, затем идёт логическое отрицание, затем вместе умножение и деление, и только потом сложение и вычитание, после них коньюнкция, потом исключающая дизъюнкция, и лишь после неё обычная дизъюнкция. В случае равных приоритетов вычисление идёт слева направо. Для изменения этого порядка выполнения используются скобки. Их следует использовать во всех сомнительных ситуациях, не перекладывая основной смысл выражения на приоритеты.

3.8 Строки

Вообще говоря, мы уже закончили рассмотрение всех тех типов, которыми ограничивался набор типов данных в автокодах и даже первых языках программирования. Но вскоре стало очевидным, что не менее важна в прикладных программах возможность обработки нечисловой информации. Сегодня, конечно, количество текстовых редакторов и процессоров, систем управления базами данных, программ-переводчиков и прочих пакетов символьной обработки существенно превосходит количество сугубо математических пакетов.

С точки зрения разработчика программного обеспечения обработка текста чрезвычайно сложна из-за разнообразия естественных языков и способов их записи. С точки зрения языков программирования обработка текста куда проще, так как подразумевается, что в языке набор символов

представляет собой короткую и упорядоченную последовательность значений. Фактически, за исключением языков с большим числом букв (восточных: китайского, японского) и языков с большим числом начертаний одной и той же буквы (семитских: арабского, мальтийского) хватает 256 значений одного байта. Последнее время всё чаще используется Уникод (Unicode) — двухбайтовая кодировка, содержащая сразу все мыслимые символы.

Строки в питоне не сильно напоминают строки в других языках программирования засчет отсутствиея типа *символ*. Обычный способ задания строкового типа состоит во введении символа и представления строк как последовательностей (массивов) символов. В питоне же символ — это строка длины 1. Таким образом, нет смысла во введении двух разных символов: обычного и уникодового, символ мы можем рассматривать как элемент соответствующей строки и не более того.

Итак, раз уж мы не можем сказать, что строка есть последовательность символова, придётся признать такое **определение**: строка есть нечто, заключённое в кавычки. Обычно используют двойные кавычки, если строка содержит одинарные внутри себя и одинарные в противном случае. Вот примеры присваивания строк:

```
a="строка"
b='ещё␣одна␣строка'
c='Он␣сказал:␣"Да"'
d="O'Хара"
```

Конечно, рано или поздно должна будет встретиться строка, содержащая оба типа кавычек. Что же делать в этом случае? Есть ещё обратные кавычки, но они уже нагружены другим смыслом, о котором чуть ниже. Поэтому используется так называемая маскировка — перед запретным символом ставится обратный слэш:

```
e='"Isnt\'t␣it?"␣-␣she␣asked.'
f="\"It␣is\"␣-␣he␣replied."
```

Видно, что маскировка — это не только средство разрешения конфликтов между кавычками, но и просто удобный в некоторых ситуациях механизм. Вообще, программист не сильно связан в этом вопросе и может по своему усмотрению решать, когда и какие кавычки использовать.

Маскировка — одна из возможностей использования управляющих последовательностей, о которых мы уже говорили раньше, при обсуждении оператора вывода и перехода на новую строку.

Если одиночный обратный слэш завершает строчку, на следующей строчке будет ожидаться продолжение строки, но сам переход записан не будет. В этой фразе ярко проявляется несовершенство терминологии. Строки (strings) — это переменные, содержащие символьные данные, которые мы обсуждаем в этой теме. Строчки (lines) — это строки экрана, по которым идут знакоместа в текстовом режиме и символы текущим шрифтом в графическом. Мы надеемся, что в каждом конкретном случае смысл будет ясен из контекста.

Переход на новую строчку можно включать в строки так же, как мы это делали для оператора `print`: как \n. Но есть ещё один способ включить

переход на новую строчку в явном виде. Для этого есть ещё два варианта заключения в кавычки. Нет, это не широко известные обратные, до которых мы всё ещё не дошли, а особые, существующие только в питоне: тройные. Они бывают двух типов: тройные двойные и тройные одинарные (тройные одинарные были введены сугубо ради симметрии и почти не используются). Выглядят они так:

```
g="""Строка,
␣явным␣образом
разбитая␣на
строчки"""
```

Ну и, наконец, перейдём к обратным кавычкам, потому как это последний их тип.

```
h=53
k=‘h‘
```

Что после этого находится в `k`? Строка, конечно. А что в строке? Значение переменной `h`, то есть `"53"`. Итак, обратные кавычки *возвращают преобразованное в строку значение переменной, имя которой записано между ними.* Другой, менее экзотичный способ совершения того же действия выглядит так:

```
k=repr(h)
```

Согласитесь, длиннее, труднее для запоминания и менее красиво. Другая от частого употребления ставшая стандартной операция со строками — это взятие её длины:

```
l=len(k)
```

Упражнение. Попробуйте угадать, что будет в `m`, `n` и `p` после выполнения следующего:

```
m,n=3.1415,"""\\\\\\\"\"␣’’’␣"""␣’’’␣"""␣’’’
p=len(‘n‘+’m’+’n’+‘m‘)
```

Правильный ответ: 3.1415, `\\\""␣’’’␣␣"""` и 43 соответственно.