Python

Урок 2. Типы данных

Базовые типы данных в языке Python

На первом занятии мы уже познакомились с типами данных целое число, вещественное число, строка, список, с логическим типом и объектом None.

Тип объекта	Пример
Числа	1234, 3.1415, 3+4j, Decimal, Fraction
Строки	spam', "guido's" , b'a\x01c'
Списки	[1, [2, 'three'], 4]
Кортежи	(1,'spam', 4, 'U')
Словари	{'food': 'spam', 'taste': 'yum'}
Множества	set('abc'), {'a', 'b', 'c'}
Файлы	myfile = open('eggs', 'r')
Прочие базовые типы	None, логические значения

В этом уроке мы рассмотрим оставшиеся типы данных и функции работы с ними.

Decimal и Fraction

Decimal (десятичное число)

Основная идея чисел Decimal в том, что компьютеры должны обеспечивать арифметические операции таким образом, чтобы операции над дробными числами работали также, как арифметика, изучаемая в школе.

Числа Decimal могут быть представлены точно. Например, вещественные числа такие как 1.1 и 2.2 не имеют точного представления в виде чисел с плавающей точкой float. Конечные пользователь обычно не ожидает, что если сложить 1.1 + 2.2 — результат будет +3,3000000000000003, как это происходит с двоичной плавающей точкой.

Сумма 0.1+0.1+0.1-0.3 должна быть в точности равна нулю, однако в случае с числами float мы получим 5.551115123125783e-17, близкое к 0, однако в точности не равное ему. Такие неточности могут накапливаться и приводить к неприятным последствиям. По этой причине, числа Decamal предпочтительнее для бухгалтерских приложений, для которых важно четкое равенство.

В отличие от чисел float, числа Decimal имеют изменяемую пользователем точность (по умолчанию до 28 знакомест), которая может быть насколько большой, насколько это необходимо для данной задачи:

Десятичное число может быть создано из целого числа, строки, числа с плавающей запятой или кортежа.

В Python 3.3 вошёл ускоритель работы Decimal (подключается автоматически). Благодаря этому ускорителю производительность Decimal повысилась настолько, что скорость вычислений стала сопоставима с int и float.

Fraction (дробное число)

Дробное число может быть создано из пары целых чисел (числитель, знаменатель), из другого дробного числа, из другого вещественного числа или из строки.

```
class fractions.Fraction(numerator=0, denominator=1)
class fractions.Fraction(other_fraction)
class fractions.Fraction(float)
class fractions.Fraction(decimal)
class fractions.Fraction(string)
```

Кортежи

Кортеж — неизменяемый список.

Зачем нужны кортежи?

1. Кортеж защищен от изменений. Поэтому его удобно использовать в случаях, когда важно быть уверенным, что данные не именялись в ходе выполнения программы.

2. Кортежи занимают меньше памяти:

```
>>> a = (1,2,3,4,5,6)
>>> b = [1,2,3,4,5,6]
>>> a.__sizeof__()
72
>>> b.__sizeof__()
88
>>>
```

3. Кортежи можно использовать в качестве ключей словаря:

```
>>> a = {(1,2,3): 100}
>>> a
{(1, 2, 3): 100}
>>> b = {[1,2,3]: 100}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

Работа с кортежами

Для кортежей можно применять все те же операции, что и для списков, за исплючением операций, пытающихся изменить кортеж.

```
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> a = (1,2) + (3,4)
>>> a
(1, 2, 3, 4)
>>> a[0]
1
>>> a[-1], a[-3:-1]
(4, (2, 3))
>>> a += (3,4)
>>> a.count(3)
2
```

```
# вызовет ошибку, т.к. такого метода у кортежа нет:
e.append(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Словари

Словарь — неупорядоченная коллекция произвольных объектов с доступом по ключу.

Создать словарь можно несколькими способами.

1. С помощью литерала словаря:

```
>>> a = {}
>>> a
{}
>>> b = {'key1' : 'value1', 'key2': 2, 'key3': [1,2,3]}
>>> b
{'key1': 'value1', 'key3': [1, 2, 3], 'key2': 2}
```

2. С помощью функции dict:

```
>>> c = dict(key='val', spam='eggs')
>>> c
{'key': 'val', 'spam': 'eggs'}
>>> d = dict([(1, 1), (2, 4)])
>>> d
{1: 1, 2: 4}
>>> d = dict([(1, 10), (2, 20)])
>>> d
{1: 10, 2: 20}
```

3. С помощью метода fromkeys:

```
>>> e = dict.fromkeys(['a', 'b', 'c'])
>>> e
{'c': None, 'b': None, 'a': None}
>>> f = dict.fromkeys(['a', 'b', 'c'], 'initial')
>>> f
{'c': 'initial', 'b': 'initial', 'a': 'initial'}
```

4. С помощью генератора словарей (см. раздел Генераторы ниже):

```
>>> d = {i : i+i for i in range(7)}
>>> d
{0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12}
```

Операции со словарями

Операция	Результат
len(d)	число элементов в словаре
d[key]	возвращает элемент словаря с ключем key
d[key] = value	присваивает значение элементу словаря
del d[key]	удаляет элемент по ключу; вызывает KeyError, если ключа нет
key in d	возвращает True если у словаря есть ключ key
key not in d	эквивалентно not key in d
dict.**fromkeys**(seq[, value])	создает новый словарь с ключами из seq и значением value
d.**get**(key[, default])	возвращает элемент словаря с ключем key, если ключ есть в словаре, иначе default; есил default не задан возвращается None, таким образом .get() никогда не вызывает исключение KeyError
d.**items**()	возвращает список пар (ключ, значение); удобно использовать для цикла по словарю
d.**keys**()	возвращает список ключей словаря

Операция	Результат
d.**pop**(key[, default])	если в словаре есть элемент с ключем key — удаляет его и возвращает его значение, в противном случае возвращает default; если default не задан — вызывается исключение KeyError
d.**popitem**()	удаляет и возвращает пару (ключ, значение); если словарь пуст, возбуждает исключение KeyError; помните, что словари неупорядочены
d.**setdefault**(key[, default])	если ключ key есть в словаре — возвращает его значение; если нет — создает ключ со значением из default и возвращает default; если default не задан используется None
d.**update**([other])	обновляет словарь, добавляя пары (ключ, значение) из other; существующие ключи перезаписываются; возвращает None (не новый словарь!)
d.**values**()	возвращает копию списка значений словаря
d.**copy**()	возвращает копию словаря
d.**crear**()	удаляет все элементы из словаря

Множества

Множество — «контейнер», содержащий не повторяющиеся элементы в случайном порядке.

```
>>> a = set()
>>> a
set()
>>> b = set(['a', 'b', 'c', 'c', 'a'])
>>> b
{'c', 'b', 'a'}
>>> c = set('hello')
>>> c
{'h', 'o', 'e', 'l'}
>>> d = {'a', 'b', 'c', 'd'}
>>> d
{'c', 'b', 'a', 'd'}
>>> e = {i ** 2 for i in range(10)} # генератор множеств
>>> e
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> f = {} # А так получится словарь
>>> type(f)
<class 'dict'>
```

Операции с множествами

Операция	Эквивалент	Результат
len(*s*)		число элементов в множестве
x in s		принадлежит ли x множеству s
s.**isdisjoint**(*t*)		истина, если set и other не имеют общих элементов
	s == t	все элементы set принадлежат other, все элементы other принадлежат set
s.**issubset**(*t*)	s <= t	все элементы set входят в other
s.**issuperset**(*t*)	s >= t	все элементы other входят в set
s.**union**(*t*, …)	s | t	объединение множеств
s.**intersection**(*t*, …)	s & t	пересечение множеств
s.**difference**(*t*, …)	s - t	вычитание множеств. возвращает множество из всех элементов set, не принадлежащие ни одному из other

Операция	Эквивалент	Результат
s.**symmetric_difference**(*t*)	s ^ t	возвращает множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих
s.**copy**()		копия множества

Методы, изменяющие множества

Метод	Эквивалент	Результат
s.**update**(*other*, …)	s |= t	объединение
s.**intersection_update**(*t*)	s &= t	пересечение
s.**difference_update**(*t*)	s -= t	вычитание
s.**symmetric_difference_update**(*t*)	s ^= t	возвращает множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих
s.**add**(*elem*)		добавляет элемент в множество
s.**remove**(*elem*)		удаляет элемент из множества; вызывает KeyError, если такого элемента не существует
s.**discard**(*elem*)		удаляет элемент, если он находится в множестве
s.**pop**()		удаляет и возвращает произвольный элемент из множества; вызывает KeyError, если множество пустое
s.**clear**()		очистка множества

Frozenset

Frozenset — полностью похож на set, но является неизменяемым типом данных. Аналогия — списки и кортежи.

Файлы

Рассмотрим встроенные средства python для работы с файлами:
открытие, закрытие, чтение и запись.

Прежде всего, необходимо открыть файл. Для этого существует встроенная функция open:

```
f = open('text.txt', 'r')
```

Первый аргумент функции — имя файла. Путь к файлу может быть относительным или абсолютным.

Второй аргумент, это режим, в котором мы будем открывать файл.

Режим	Описание
'r'	открытие на чтение (является значением по умолчанию)
w'	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый
'a'	открытие на дозапись, информация добавляется в конец файла
'b'	открытие в двоичном режиме
't'	открытие в текстовом режиме (является значением по умолчанию)
'+'	открытие на чтение и запись

Режимы могут быть объединены, то есть, к примеру, ‘rb’ — чтение в двоичном режиме. По умолчанию режим равен ‘rt’.

И последний аргумент, encoding, нужен только в текстовом режиме чтения файла. Этот аргумент задает кодировку.

Методы работы с файлами

Метод	Описание
f.read()	Чтение файла целиком в единственную строку

Метод	Описание
f.read(N)	Чтение следующих N символов (или байтов) в строку
f.readline()	Чтение следующей текстовой строки (включая символ конца строки) в строку
f.readlines()	Чтение файла целиком в список строк (включая символ конца строки)
f.write(string)	Запись строки символов (или байтов) в файл
f.writelines(list)	Запись всех строк из списка в файл
f.close()	Закрытие файла вручную (выполняется по окончании работы с файлом)
f.flush()	Выталкивает выходные буферы на диск, файл остается открытым
f.seek(N)	Изменяет текущую позицию в файле для следующей операции, смещая ее на N байтов от начала файла.
for line in open('data'): *операции над line*	Итерации по файлу, построчное чтение
open('f.txt', encoding='latin-1')	Файлы с текстом Юникода в Python 3.0
open('f.bin', 'rb')	Чтение двоичных файлов

Генераторы

Список

```
>>> l = [i for i in range(10)]
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Кортеж

Генератор кортежа вовращает не объект кортеж, а объект-генератор, что позволяет существенно экономить занимаемую кортежем память:

```
>>> t = (i for i in range(10))
>>> t
<generator object <genexpr> at 0x106d05cd0>
>>> t.__sizeof__()
48
>>> l.__sizeof__()
168
```

Словарь

```
>>> d = {i : i+i for i in range(7)}
>>> d
{0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12}
```

Множество

```
>>> s = {i for i in range(10)}
>>> s
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Генераторы особенно удобно использовать со встроенными функциями, такими как sum(), min(), max():

```
# подсчитаем максимальную длину строки в файле
>>> max(len(line) for line in open('text.txt') if line.strip())
79
```