Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» и профилизации "Искусственный интеллект и математическая экономика" ММФ БГУ

Тема 2. Объект как фундаментальное понятие в Python. Встроенные типы объектов

доц. Лаврова О.А., доц. Щеглова Н.Л., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

февраль, 2025

2.1 Понятие объекта. Атрибуты. Методы. Встроенные типы объектов

Объект в Python — это базовая сущность для представления данных.

К объектам в Python относятся: данные встроенных типов, встроенные и пользовательские функции, файлы, модули, встроенные и пользовательские классы, экземпляры классов и т.д.

Внутренне объекты представляют собой области выделенной памяти с достаточным пространством для хранения информации об объекте.

У объекта могут быть *метаданные*, которые называются атрибутами. **Атрибут** — это имя переменной или имя функции, которое связано с объектом и используется для получения информации об объекте и совершения действий над этим объектом. Все атрибуты объекта составляют *пространство имен объекта*.

Доступ к атрибуту `attr` объекта `obj` производится с применением синтаксиса уточнения `obj.attr`. Доступ к атрибутам объекта является одним из самых распространенных выражений в Python

In [5]:
```
list1 = [1,2];
list1.__class__
```

Out[5]:
```
list
```

Атрибуты объектов могут быть именами переменных или именами функций. Если атрибут объекта является именем функции, то он называется **методом**. Для вызова

функции следует указать ее имя, затем в круглых скобках через запятую перечислить аргументы вызова функции

In [8]:
```
list1.append(3); print(list1)
list1.__class__ # атрибут, не являющийся методом
```

```
[1, 2, 3]
```

Out[8]:
```
list
```

Просмотреть пространство имен объекта можно с помощью встроенной функции `dir`. Аргументом функции `dir` является экземпляр объекта или тип объекта

In [11]:
```
print(dir(list1))
dir(list1) == dir(list)
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__d
elitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribut
e__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__iadd__', '__imul_
_', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rev
ersed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__s
ubclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
```

Out[11]:
```
True
```

Атрибуты, которые начинаются и заканчиваются двумя знаками подчеркивания, *чаще всего* являются **методами, реализующими операции** над объектами (*dunder method*, dunder comes from 'double underscore' или *magic methods*)

In [14]:
```
print([attr for attr in dir(list1) if attr.startswith('_')])
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__d
elitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribut
e__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__iadd__', '__imul_
_', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rev
ersed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__s
ubclasshook__']
```

Операция (operator) — это конструкция языка, аналогичная по записи математическим операциям. Для обозначения операции часто используют не алфавитно-цифровые, а специальные символы.

In [17]:
```
a = 1
a + 2, a.__add__(2)
```

Out[17]:
```
(3, 3)
```

Встроенные типы объектов

Встроенные типы объектов делятся на **числа** и **коллекции**.

Типы объектов, представляющих числа: **целые числа (`int`)**, **числа с плавающей точкой (`float`)**, **комплексные числа (`complex`)**. **Булевый тип (`bool`)**

наследуется от типа `int`, поэтому булевый тип относится к типам числовых объектов.

Коллекции — это наборы объектов, которые делятся на три вида:

Последовательности — это *позиционно упорядоченные* коллекции. Встроенные типы, относящиеся к последовательностям: **строки (`str`)**, **списки (`list`)**, **кортежи (`tuple`)**.

Отображения — это неупорядоченные коллекции объектов, *хранящихся по ключам*. Встроенные типы, относящиеся к отображениям: **словарь (`dict`)**.

Множества — это неупорядоченные коллекции *не повторяющихся* объектов. Встроенные типы, относящиеся к множествам: **множество (`set`)**.

Встроенная функция `type(obj)` возвращает тип объекта `obj`

In [23]:
```
type(1), type('1'), type([1])
```

Out[23]:
```
(int, str, list)
```

Встроенные типы являются частью базового языка Python и всегда доступны для использования.

Встроенные типы Python задействуют уже оптимизированные алгоритмы для работы со структурами данных, которые реализованы на C, чтобы обеспечивать высокое быстродействие.

Встроенные типы объектов не только облегчают программирование, но также являются более мощными и эффективными структурами данных, чем большинство того, что можно создать в Python самостоятельно.

Создание объектов встроенных типов

Создавать объекты можно с помощью *литералов* и с помощью *функций-конструкторов* объектов.

Литерал в Python — это форма записи данных для их представления объектом определеннного типа. Синтаксис литерала определяет тип создаваемого объекта. Приведем пример, как от различных форм записи числа 1, зависит тип создаваемого объекта:

In [28]:
```
print([type(obj) for obj in [1, 1., 1j, '1', [1], (1,), {1}, {1:1}]])
```

```
[<class 'int'>, <class 'float'>, <class 'complex'>, <class 'str'>, <class 'lis
t'>, <class 'tuple'>, <class 'set'>, <class 'dict'>]
```

Имя **функции-конструктора** совпадает с названием типа создаваемого объекта

In [30]:
```
int(), float(), complex(), str(), list(), tuple(), dict(), set([1])
```

Out[30]: `(0, 0.0, 0j, '', [], (), {}, {1})`

Выражение (expression) в Python — это конструкция языка в виде комбинации литералов, переменных, операций и вызовов функций, которая вычисляется и возвращает значение в виде объекта. Выражение не хранится в памяти, в памяти сохраняется результирующий объект.

2.2 Встроенные типы объектов, представляющих числа

Примеры создания целых чисел (`int`) в различных системах счисления:

In [35]:
```
1_000_111_111_000, 0b11, 0o11, 0x11, int()
```

Out[35]: `(1000111111000, 3, 9, 17, 0)`

Примеры создания чисел с плавающей точкой (`float`):

In [37]:
```
3.1415, 3e-10, 1E-1, float(2)
```

Out[37]: `(3.1415, 3e-10, 0.1, 2.0)`

С помощью символов `e` или `E` задается экспоненциальная форма записи чисел с плавающей точкой.

Тип `float` является аналогом 64-битного *double* в C/C++.

Числа с плавающей точкой (`float`) не способны представлять определенные значения точно из-за ограниченного количества битов. Это фундаментальная проблема в численном программировании, не уникальная для языка Python

In [43]:
```
1.1 + 2.2 == 3.3, 1.1 + 2.2 - 3.3
```

Out[43]: `(False, 4.440892098500626e-16)`

Примеры создания комплексных чисел (`complex`):

In [45]:
```
3+4j, 1J, complex(1,1)
```

Out[45]: `((3+4j), 1j, (1+1j))`

Мнимая часть комлексных чисел заканчивается символом `j` или `J` .

Действительная и мнимая части комплексного числа доступны через атрибуты `real` и `imag` , соответственно

In [48]:
```
c = 1J; c.real, c.imag
```

Out[48]: `(0.0, 1.0)`

Некоторые арифметические операции

Операция (operator) — это конструкция языка, аналогичная по записи математическим операциям. Для обозначения операции часто используют не алфавитно-цифровые, а специальные символы.

Операция возведения в степень (`**`) реализована для всех числовых типов

In [52]:
```
3**456, 15.**2., (1 + 15.J)**2
```

Out[52]:
```
(3692258952335548868486253430960682864635485853933963934010870460686027831695461222301192671449678638056692404147215681314705828383718606775942559434139077262724892563647389406590090689494677802027036224351161724135952 1,
 225.0,
 (-224+30j))
```

Встроенная функция `pow(x,y)` также позволяет возводить число `x` в степень `y`

In [56]:
```
pow(3,456), pow(15.,2.), pow(1 + 15.J,2)
```

Out[56]:
```
(3692258952335548868486253430960682864635485853933963934010870460686027831695461222301192671449678638056692404147215681314705828383718606775942559434139077262724892563647389406590090689494677802027036224351161724135952 1,
 225.0,
 (-224+30j))
```

Операция целочисленного деления с округлением в меньшую сторону (`//`) реализована для типов `int` и `float`

In [58]:
```
15//2, 15.2//2
```

Out[58]:
```
(7, 7.0)
```

Операция вычисления остатка от деления (`%`) реализована для типов `int` и `float`

In [60]:
```
15%2, 15.2%2
```

Out[60]:
```
(1, 1.1999999999999993)
```

In [61]:
```
15/2, 15//2, 15%2
```

Out[61]:
```
(7.5, 7, 1)
```

Если для операций сложения (`+`), вычитания (`-`) или умножения (`*`) в качестве операндов указаны числовые объекты РАЗНЫХ типов, то процесс вычисления следующий. Сначала операнды преобразуются к типу, который представляет наиболее широкое математическое множество, а затем выполняется операция над операндами одинакового типа

In [63]:
```
(23 + 1.2 + 1J)**2
```

Out[63]: (584.64+48.4j)

Некоторые встроенные функции для работы с числами

Функции преобразования целого числа (`int`) из десятичной системы счисления в двоичную `bin` , в восьмеричную `oct` , в шестнадцатеричную `hex` системы счисления:

In [66]:
```
bin(3), oct(8), hex(15)
```

Out[66]: ('0b11', '0o10', '0xf')

Функции преобразования `bin` , `oct` , `hex` возвращают строковый объект

In [68]:
```
?abs
```

```
Signature: abs(x, /)
Docstring: Return the absolute value of the argument.
Type:      builtin_function_or_method
```

In [69]:
```
abs(-1), abs(complex(1,1))
```

Out[69]: (1, 1.4142135623730951)

In [70]:
```
?round
```

```
Signature: round(number, ndigits=None)
Docstring:
Round a number to a given precision in decimal digits.

The return value is an integer if ndigits is omitted or None.  Otherwise
the return value has the same type as the number.  ndigits may be negative.
Type:      builtin_function_or_method
```

In [71]:
```
round(10.51), round(10.555555, 4)
```

Out[71]: (11, 10.5556)

Модули и расширения для работы с числами

Для работы с типом десятичных чисел (`decimal`) необходимо импортировать модуль `decimal` . **Десятичные числа (`decimal`)** являются числами с плавающей точкой с *фиксированным количеством значащих цифр*. Вычисления с числами типа `decimal` могут иметь лучшую точность, чем вычисления с числами типа `float` .

In [78]:
```
import decimal as dec
print(0.1+0.2, dec.Decimal('0.1')+dec.Decimal('0.2'))
```

```
0.30000000000000004 0.3
```

Для работы с типом рациональных чисел (`fraction`) необходимо импортировать модуль `fractions` . **Рациональные числа (`fraction`)** явно хранят числитель и знаменатель, что позволяет избежать ряда неточностей и ограничений в вычислениях с числами с плавающей точкой (`float` , `decimal`).

Невстроенные типы числовых объектов `decimal` и `fraction` предлагают способы получения более точных результатов вычислений (например, необходимые в финансовых приложениях), хотя ценой снижения скорости вычислений и более многословного кода.

Более сложные инструменты для работы с числами содержатся в модуле `math`.

Модуль `cmath` предназначен для работы с комплексными числами.

Модуль `random` выполняет генерацию случайных чисел и случайный выбор.

Расширение `numpy` предоставляет инструменты для эффективного численного программирования на основе специального типа данных массив `ndarray`.

2.3 Встроенные типы последовательностей: str, list, tuple

Последовательность — это *позиционно упорядоченная* коллекция объектов.

Строка (`str`) — это последовательность символов для хранения и представления текста, которую *нельзя изменять* после ее создания (read-only, immutable).

Встроенный тип объектов-символов не определен в Python (отсутствует аналог, например, типа `char` в C, C++).

Список (`list`) — это последовательность объектов произвольных типов, которую *можно изменять* после ее создания.

Кортежи (`tuple`) — это последовательность объектов произвольных типов, которую *нельзя изменять* после ее создания (read-only, immutable).

Создание последовательностей

Литерал, который создает объект типа список имеет вид `[elem1, elem2,...,elemN]`, где элементами списка могут быть объекты Python произвольного типа. Создание списка также возможно с помощью функции-конструктора `list()`

In [89]:
```
[1, [1+2j, [4e1]], 4.5], list(range(3))
```

Out[89]:
```
([1, [(1+2j), [40.0]], 4.5], [0, 1, 2])
```

Квадратные скобки `[...]` используются также для отображения объектов типа список.

Список поддерживает произвольное вложение объектов.

Литерал, который создает объект типа кортеж имеет вид `(elem1, elem2,...,elemN)`, где элементами кортежа могут быть объекты Python произвольного типа. Создание кортежа также возможно с помощью фукнции-конструктора `tuple()`

In [93]:
```
(),(1, ([2], (6,7)), 4.5), tuple([1,20,3])
```

Out[93]:
```
((), (1, ([2], (6, 7)), 4.5), (1, 20, 3))
```

Круглые скобки используются также для отображения объектов типа кортеж.

Кортеж поддерживает произвольное вложение объектов.

При создании одноэлементных кортежей необходимо обязательно указывать запятую после элемента

In [97]:
```
(1,)
```

Out[97]:
```
(1,)
```

Круглые скобки, окружающие элементы кортежа, часто можно опускать; запятые — это то, что фактически определяет кортеж

In [99]:
```
1, 2, 3
```

Out[99]:
```
(1, 2, 3)
```

Строковые объекты имеют много способов их записи в коде. Строки в Python разрешено заключать в 'одинарные' или "двойные" кавычки. Различные кавычки обозначают то же самое, но позволяют встраивать в строковый объект кавычки другого вида. Создание строки также возможно с помощью фукнции-конструктора `str()`

In [101…
```
'spa"m', "Bob's", str(1)
```

Out[101…
```
('spa"m', "Bob's", '1')
```

Для читабельности кода при создании строк предпочтение отдается одинарным кавычкам.

Пустая строка задается парой кавычек (одинарных или двойных), между которыми ничего нет или вызовом функции-конструктора `str` без аргументов:

In [111…
```
'', str()
```

Out[111…
```
('', '')
```

Многострочные строковые литералы задаются в Python с помощью утроенных кавычек

In [113…
```
"""
1
2
"""
```

Out[113…
```
'\n1\n2\n'
```

Текст между утроенными кавычками собирается в единственную строку с дополнительными символами новой строки `\n` в местах, где в коде присутствуют разрывы строк. Строки в утроенных кавычках часто применяются для задания **строк документации** для объектов Python.

В строках может использоваться управляющий символ обратной косой черты `\` для введения специальных символов, управляющих отображением содержимого строки и которые не могут быть легко набраны на клавиатуре. Символ \ и один или более следующих за ним символов в строковом литерале в результирующем строковом объекте заменяются символом, который имеет значение, указанное управляющей последовательностью.

Например, два символа `\n` обозначают переход на новую строку, `\t` заменяется символом табуляции

In [116…
```
print('1 \t 2 \n 3')
```

```
1	 2
 3
```

Для явного представления символа обратной косой черты в строке его необходимо удваивать `\\`

In [118…
```
print('\\')
```

```
\
```

Если Python не распознает символ после \ в качестве допустимого управляющего кода, то он просто оставляет обратную косую черту в результирующей строке

In [121…
```
print('\c \xc6')
```

```
\c Æ
<>:1: SyntaxWarning: invalid escape sequence '\c'
<>:1: SyntaxWarning: invalid escape sequence '\c'
C:\Users\Olga\AppData\Local\Temp\ipykernel_14068\1155648579.py:1: SyntaxWarning:
invalid escape sequence '\c'
  print('\c \xc6')
```

Для создания строк, отменяющих действие управляющих символов обратной косой черты, используется символ `r` или `R` перед открывающей кавычкой строки. В результате Python сохраняет управляющие символы так, как они набраны

In [123…
```
print('1\n2\t\t3\n', r'1\n2\t\t3\n')
```

```
1
2               3
 1\n2\t\t3\n
```

Для создания байтовых строк используется символ `b` или `B` перед открывающей кавычкой строки

Байтовая строка (bytes) — это последовательность символов для хранения и представления *байтовой* информации, используемой в файлах медиа-данных и при передаче через сеть.

Для создания строк Unicode используется символ `u` или `U` перед открывающей кавычкой строки

In [125…
```
b'a\x01c', u'sp\xc4m'
```

Out[125…
```
(b'a\x01c', 'spÄm')
```

Строки в Python имеют фиксированные размеры и не могут быть изменены после создания!

Операция индексации `[]`

Последовательности поддерживают операцию индексации, или доступа к элементам по смещению. Индексация элементов последовательностей начинается с нуля. Для доступа к элементу последовательности индекс элемента указывается в квадратных скобках `X[i]`, что означает предоставить из последовательности X содержимое для позиции `i`.

In [129…
```
list1 = [2,3.3,'str']; list1[1]
```

Out[129…
```
3.3
```

In [130…
```
str1 = 'String'; str1[0]
```

Out[130…
```
'S'
```

In [131…
```
tuple1 = (1, str1, list1); tuple1[1]
```

Out[131…
```
'String'
```

Обратите внимание, что результатом индексации элементов строки всегда является строка

In [133…
```
str1[0][0], str1[0][0][0]
```

Out[133…
```
('S', 'S')
```

Возможна индексация элементов последовательности отрицательными значениями. В этом случае доступ к элементам последовательности осуществляется с конца

In [135…
```
str1, str1[-1]
```

Out[135…
```
('String', 'g')
```

In [136…
```
list1, list1[-2]
```

Out[136…
```
([2, 3.3, 'str'], 3.3)
```

In [137…
```
tuple1, tuple1[-3]
```

Out[137…
```
((1, 'String', [2, 3.3, 'str']), 1)
```

Индексация за пределами последовательности приводит к ошибке

In [139…
```
str1
```

Out[139…
```
'String'
```

In [140…
```
str1[10]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[140], line 1
----> 1 str1[10]

IndexError: string index out of range
```

В дополнение к позиционной индексации элементов последовательности в Python поддерживается обобщенная форма индексации — нарезание. **Нарезание** представляет собой способ извлечения ЧАСТИ последовательности или СРЕЗА за один раз.

Срез `X[i:j]` означает "предоставить из последовательности `X` все содержимое, начиная с позиции `i` и заканчивая позицией `j`, *не включая содержимое для позиции* `j`".

In [ ]:
```
str1, str1[1:3], str1[2:-1]
```

Срез возвращает новый объект последовательности.

Если левая позиция среза не указана `X[:j]`, тогда по умолчанию левая позиция равна 0. Если правая позиция среза не указана `X[i:]`, тогда по умолчанию правая позиция равна количеству элементов нарезаемой последовательности.

In [ ]:
```
str1, str1[1:], str1[:3], str1[:]
```

Срез за границами последовательности не приводит к ошибке, так как позиции, выходящие за границы последовательности, корректируются

In [ ]:
```
str1, str1[-100:100], str1[10:]
```

Результатом индексации элементов последовательности `X[i:j]` в обратном порядке, когда нижняя граница больше верхней `i>j`, является пустая

последовательность:

In [ ]:
```
str1[2:1], list1[3:1], tuple[4:1]
```

В Python поддерживается также **расширенное нарезание** `X[i:j:k]` с шагом `k`, значение которого по умолчанию равно `1`:

In [ ]:
```
str1, str1[::2], str1[1::3], str1[::-1]
```

Операция индексации `[...]` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция индексации `[]` реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`).

Операция `+` для последовательностей реализует их конкатенацию

In [ ]:
```
str1, str1 + str1
```

In [ ]:
```
list1, list1 + list1
```

In [ ]:
```
tuple1, tuple1 + tuple1
```

В выражении с операцией сложения `+` для последовательностей в качестве операндов могут быть использованы только объекты одного и того же типа. Нельзя использовать последовательности различных типов в операции сложения `+` из-за **строгой типизации объектов** в Python. Автоматическое преобразование типов не производится в Python.

In [ ]:
```
[1] + '1'
```

Следует отметить, что для числовых объектов при выполнении операции `+` допускается смешение типов операндов:

In [ ]:
```
1+1.1
```

Операция сложения `+` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция сложения `+` определена в Python, где в качестве обоих операндов могут испольоваться

- числовые объекты (`int` , `float` , `complex`) или
- строки (`str`) или
- списки (`list`) или
- кортежи (`tuple`).

Операция `*` для последовательности реализует ее повторение

В операции `*` одним из операндов является последовательность, вторым -- целое число, которое определяет число повторений.

In [142…
```
str1, str1*4
```

Out[142…
```
('String', 'StringStringStringString')
```

In [143…
```
list1, list1*2
```

Out[143…
```
([2, 3.3, 'str'], [2, 3.3, 'str', 2, 3.3, 'str'])
```

In [144…
```
tuple1, 0*tuple1
```

Out[144…
```
((1, 'String', [2, 3.3, 'str']), ())
```

Операция умножения `*` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция умножения `*` определена в Python, где в качестве обоих операндов могут ипользоваться

- числовые объекты (`int` , `float` , `complex`) или
- строка (`str`) и целое число или
- список (`list`) и целое число или
- кортеж (`tuple`) и целое число.

Операция `len` вычисления числа элементов

In [147…
```
str1 = 'String'; len(str1)
```

Out[147…
```
6
```

In [148…
```
list1 = range(10); len(list1)
```

Out[148…
```
10
```

In [149…
```
tuple1 = (1, str1, list1); len(tuple1)
```

Out[149…
```
3
```

Операция `len(obj)` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `len(obj)` реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`).

Операция `in` проверки принадлежности

Операция `in` (`not in`) проверки принадлежности (не принадлежности) возвращает булевый объект `True` или `False`.

Операция `elem in sequence` проверяет принадлежность элемента `elem` последовательности `sequence`

In [154…
```
str1 = 'String'; 't' in str1
```

Out[154…
```
True
```

In [155…
```
list1 = range(10); 10 not in list1
```

Out[155…
```
True
```

In [156…
```
tuple1 = (1, str1, list1); 2 in tuple1
```

Out[156…
```
False
```

Операция `substring in string` проверяет принадлежность подстроки `substring` строковому объекту `string`

In [158…
```
str1 = 'String'; 'ing' in str1
```

Out[158…
```
True
```

Операция `in` (`not in`) может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `in` (`not in`) реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)

Встроенные функции для работы со строками

Функция `ord(s)` преобразовывает односимвольную строку `s` в целочисленный код символа в кодировочной таблице

In [162…
```
?ord
```

```
Signature: ord(c, /)
Docstring: Return the Unicode code point for a one-character string.
Type:      builtin_function_or_method
```

In [164…
```
ord('й')
```

Out[164…
```
1081
```

Функция `chr(code)` выполняет обратное преобразование, принимая целочисленный код символа `code` и возвращая символ, соответствующий коду

In [167…
```
?chr
```

```
Signature: chr(i, /)
Docstring: Return a Unicode string of one character with ordinal i; 0 <= i <= 0x1
0ffff.
Type:      builtin_function_or_method
```

In [168…
```
chr(1081)
```

Out[168…
```
'й'
```

Модуль стандартной библиотеки `re` предназначен для обработки строк на основе образцов.

2.4 Встроенный тип отображения: dict

Отображение — это неупорядоченная коллекция элементов в виде пары объектов *ключ: значение*.

Словарь (`dict`) — это единственный встроенный тип Python, который относится к отображениям.

Создание словаря

Литерал, который создает объект типа словарь имеет вид `{key1: value1, key2: value2,...,keyN: valueN}` , элементы словаря задаются парами объектов `ключ: значение`

In [175…
```
{1: 1, 'second': [2], (3,): {3:3}}
```

Out[175…
```
{1: 1, 'second': [2], (3,): {3: 3}}
```

Ключами словаря могуть быть только числа (`int` , `float` , `complex`), строки(`str`), кортежи (`tuple`). Значениями словаря могут быть объекты произвольных типов. Словарь поддерживает произвольное вложение объектов.

Каждый ключ словаря способен иметь только одно ассоциированное значение

In [179…
```
{1:1, 1:2, 1:3}
```

Out[179…
```
{1: 3}
```

Одинаковое значение может храниться под любым количеством ключей в словаре

In [181…
```
{1:1, 2:1, 3:1}
```

Out[181…
```
{1: 1, 2: 1, 3: 1}
```

Создать словарь также можно с помощью функции-конструктора `dict`

In [184…
```
dict(one=1, two=2), dict(zip(['one', 'two'], [1, 2]))
```

Out[184…
```
({'one': 1, 'two': 2}, {'one': 1, 'two': 2})
```

In [187…
```
?zip
```

```
Init signature: zip(self, /, *args, **kwargs)
Docstring:
zip(*iterables, strict=False) --> Yield tuples until an input is exhausted.

   >>> list(zip('abcdefg', range(3), range(4)))
   [('a', 0, 0), ('b', 1, 1), ('c', 2, 2)]

The zip object yields n-length tuples, where n is the number of iterables
passed as positional arguments to zip().  The i-th element in every tuple
comes from the i-th iterable argument to zip().  This continues until the
shortest argument is exhausted.

If strict is true and one of the arguments is exhausted before the others,
raise a ValueError.
Type:           type
Subclasses:
```

Операция индексации `[]`

Словари не поддерживают позиционное упорядочение слева направо, как последовательности. Обращение к значению словаря/индексация словаря происходит по ключу.

Внутренне словари реализованы как хеш-таблицы, обеспечивая очень быстрое извлечение элементов.

Выражением индексации словаря по ключу являются квадратные скобки `[]`. Операция индексации позволяет извлекать и изменять значения, связанные с ключами.

In [192…
```
dict1 = {'one': 1}

dict1['one'] = 11 # изменение значения с использованием индексации
dict1['one'] # извлечение значения с использованием индексации
```

Out[192…
```
11
```

Присваивание элементу словаря, проиндексированному по несуществующему ключу, приводит к созданию НОВОГО элемента словаря с указанным ключом и указанным значением:

In [194…
```
dict1['new'] = 2
dict1
```

Out[194…
```
{'one': 11, 'new': 2}
```

При идексации по ключу, который является кортежем, круглые скобки для обозначения кортежа можно опускать:

In [196…
```
dict1[1,2] = 12
dict1
```

Out[196…
```
{'one': 11, 'new': 2, (1, 2): 12}
```

Индексация по несуществующему ключу словаря приводит к ошибке:

In [198…
```
dict1['two']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[198], line 1
----> 1 dict1['two']

KeyError: 'two'
```

Обращение к элементу словаря по несуществующему ключу с помощью словарного метода `get` не приведет к ошибке и вернет `None`:

In [199…
```
print(dict1.get('two'))
```

```
None
```

Словарь не имеет фиксированных размеров.

Операция индексации `[...]` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция индексации `[...]` реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)
- словарей (`dict`)

Операции сложения `+` и умножения `*` не реализованы для словарей

Операция (`X | Y`) реализует объединение словарей `X` и `Y`. В случае совпадения ключей в объединенном словаре сохраняется то значение, которое является крайним правым в выражении

In [204…
```
X = dict(one=1); Y = {'one':'updated', 'two':2}
X | Y
```

Out[204…
```
{'one': 'updated', 'two': 2}
```

Оператор (`X |= Y`) присваивает словарю `X` результат объединения двух словарей `X` и `Y`:

In [206…
```
X |= Y; X
```
Out[206…
```
{'one': 'updated', 'two': 2}
```

Операция `len` вычисления числа элементов

In [211…
```
dict1, len(dict1)
```
Out[211…
```
({'one': 11, 'new': 2, (1, 2): 12}, 3)
```

Операция `len(obj)` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `len(obj)` реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)
- словарей (`dict`)

Операция `in` проверки принадлежности

Операция `in` (`not in`) проверки принадлежности (не принадлежности) КЛЮЧА СЛОВАРЮ возвращает булевый объект `True` или `False` .

In [258…
```
dict1
```
Out[258…
```
{'one': 11, 'new': 2, (1, 2): 12}
```

In [259…
```
'one' in dict1, 'three' not in dict1
```
Out[259…
```
(True, True)
```

In [261…
```
dict1['one'] in dict1
```
Out[261…
```
False
```

Операция `in` (`not in`) может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `in` (`not in`) реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)
- словарей (`dict`)

Кодовые трюки со словарями

Словарные ключи в виде кортежей могут применяться для реализации *разреженных структур данных*, например, представляя таким образом многомерные массивы,

где значения хранятся лишь в нескольких позициях

In [266…
```
matrix = {}
matrix[1,10,100] = 5; matrix[100,100,100] = 500
matrix[(1,10,100)]
```

Out[266…
```
5
```

В приведенном примере ключи `(i,j,k)` представляют собой индексы элемента трехмерного массива $a_{i,j,k}$.

2.5 Встроенный тип множества: set

Множество (`set`) — это неупорядоченная коллекция не повторяющихся объектов. Элемент встречается во множестве только однажды независимо от того, сколько раз он добавлялся. Множество может содержать только числа, строки и кортежи (неизменяемые объекты). Множество не имеет фиксированных размеров.

Множества подобны словарям без значений. Так как элементы множества неупорядоченные, уникальные и неизменяемые, они ведут себя очень похоже на ключи словаря.

Литерал множества, как и литерал словаря, начинается и заканчивается фигурными скобками `{elem1, elem2, ..., elemN}` . Отличия литералов множества и словаря заключаются в представлении элементов соответствующей коллекции:

In [273…
```
{1}, {1:1}
```

Out[273…
```
({1}, {1: 1})
```

Выбор синтаксиса с фигурными скобками для литералов множеств логичен, так как множества очень похожи на ключи в словарях без значений.

Пустое множество создать с помощью литерала нельзя

In [277…
```
{}, type({})
```

Out[277…
```
({}, dict)
```

Примеры создания множеств, которые демонстрируют, что повторяющиеся элементы удаляются из множества

In [280…
```
{1,2,3,3,1}, set('setset')
```

Out[280…
```
({1, 2, 3}, {'e', 's', 't'})
```

Операции индексации `[]` , сложения `+` и умножения `*` не реализованы для множеств

Операция `len` вычисления числа элементов

In [284…
```
X = {1,1,2,3}; len(X)
```

Out[284…
```
3
```

Операция `len(obj)` может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `len(obj)` реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)
- словарей (`dict`)
- множеств (`set`)

Операция `in` проверки принадлежности

Операция `in` (`not in`) проверки принадлежности (не принадлежности) возвращает булевый объект `True` или `False` .

In [289…
```
Y = {4,5}; print(X, Y)
```

```
{1, 2, 3} {4, 5}
```

In [291…
```
2 in X, X not in Y
```

Out[291…
```
(True, True)
```

Операция `in` (`not in`) может выполняться над объектами различных типов, т.е. операция обладает **полиморфизмом** в Python. Например, операция `in` (`not in`) реализована в Python для

- строк (`str`)
- списков (`list`)
- кортежей (`tuple`)
- словарей (`dict`)
- множеств (`set`)

Операции над множествами: &, |, -, <, <=

Определим два множества:

In [296…
```
X = set(range(3)); Y = set(range(4))
X, Y
```

Out[296…
```
({0, 1, 2}, {0, 1, 2, 3})
```

Операция (`&`) реализует пересечение множеств:

In [299…
```
print(X, Y, X & Y)
```
```
{0, 1, 2} {0, 1, 2, 3} {0, 1, 2}
```

Операция (`|`) реализует объединение множеств:

In [302…
```
print(X, Y, X | Y)
```
```
{0, 1, 2} {0, 1, 2, 3} {0, 1, 2, 3}
```

Операция (`-`) реализует разность множеств:

In [305…
```
print(X, Y, X - Y, Y - X)
```
```
{0, 1, 2} {0, 1, 2, 3} set() {3}
```

Операции (`<` и `<=`) являются проверкой вложенности множеств:

In [308…
```
print(X, Y, X > Y, X < X, X <= X)
```
```
{0, 1, 2} {0, 1, 2, 3} False False True
```

Кодовые трюки с множествами

В Python существует подход для удаления повторяющихся элементов ЛЮБОЙ последовательности с помощью использования множества

In [312…
```
list1 = list(range(5))*2
print(list1)

list1 = list(set(list1))
print(list1)
```
```
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```

В Python существует подход на основе использования множеств для проверки равенства двух последовательностей, когда порядок следования элементов в последовательностях не важен

In [314…
```
set('abcdef') == set('fedcba')
```

Out[314…
```
True
```