**Тема занятия №03: Числа, строки, списки, кортежи, множества, диапазоны, словари**

**22.1 Введение**

Общая информация по множеству в Python:

**Множество в языке Питон** — это структура данных, эквивалентная множествам в математике. Множество может состоять из различных элементов, порядок элементов в множестве неопределен. В множество можно добавлять и удалять элементы, можно перебирать элементы множества, можно выполнять операции над множествами (объединение, пересечение, разность). Можно проверять принадлежность элементу множества.

В отличии от массивов, где элементы хранятся в виде последовательного списка, в множествах порядок хранения элементов не определен (более того, элементы множества хранятся не подряд, как в списке, а при помощи хитрых алгоритмов). Это позволяет выполнять операции типа “проверить принадлежность элемента множеству” быстрее, чем просто перебирая все элементы множества.

Элементами множества может быть любой неизменяемый тип данных: числа, строки, кортежи. Изменяемые типы данных не могут быть элементами множества, в частности, нельзя сделать элементом множества список (но можно сделать кортеж) или другое множество. Требование неизменяемости элементов множества накладывается особенностями представления множества в памяти компьютера.

**22.2 Работа с множествами**

Одно из основных свойств множеств заключается в уникальности каждого из их элементов. Посмотрим, что получится, если сформировать set из строчки с заведомо повторяющимися символами:

**strange\_app = set('TikTok')**

**print(strange\_app)**

**> {'o', 'T', 'i', 'k'}**

Из результата были удалены дублирующиеся в слове 'TikTok' символы. Так множества в очередной раз доказали, что содержат в себе только уникальные элементы.

Немаловажным является и тот факт, что при литеральном объявлении, итерируемые объекты сохраняют свою структуру.

**pangram = {'съешь же ещё этих мягких французских булок, да выпей чаю'}**

**print(pangram)**

**> {'съешь же ещё этих мягких французских булок, да выпей чаю'}**

Для сравнения:

**pangram\_second = set('съешь же ещё этих мягких французских булок, да выпей чаю')**

**print(pangram\_ second)**

# попить чаю с функцией set(), к сожалению, не выйдет

**> {'щ', 'ь', 'э', 'н', 'з', 'с', 'м', ' ', 'р', 'о', 'ю', 'ш', 'ё', 'к', 'у', 'е', 'л', 'в', 'г', 'ы', 'ъ', 'х', 'ж', 'ц', 'п', 'и', ',', 'ч', 'а', 'т', 'й', 'ф', 'д', 'я', 'б'}**

Отдельное python множество может включать в себя объекты разных типов:

**we\_are\_the\_py\_objects = {None, True, 42, 3.14, 'Force', ('Dark Side', 'Light Side')}**

**print(we\_are\_the\_py\_objects)**

**> {True, 3.14, ('Dark Side', 'Light Side'), 42, 'Force', None}**

Здесь нет никакого противоречия с математической дефиницией, так как все составляющие we\_are\_the\_py\_objects имеют вполне конкретное общее свойство, являясь объектами языка Питон.

Но не стоит забывать и внутреннее определение set-ов. Важно помнить, что list-ы и dict-ы не подходят на роль элементов множества, из-за своей изменяемой природы.

**glados = {['Great cake']}**

**print(glados)**

**> Traceback (most recent call last):**

**glados = {['Great cake']}**

**TypeError: unhashable type: 'list'**

Функция set(), тем не менее, корректно обрабатывает случаи, когда ей на вход подаются списки или словари.

**# словарь будет преобразован во множество его ключей, значения отбрасываются**

**some\_dict = {'key\_one': 'val\_one', 'key\_two': 'val\_two'}**

**some\_set = set(some\_dict)**

**print(some\_set)**

**> {'key\_one', 'key\_two'}**

**# элементы списка преобразуются в элементы множества, дубликаты удаляются**

**card\_suit = ['heart', 'diamond', 'club', 'spade', 'spade']**

**suit\_set = set(card\_suit)**

**print(suit\_set)**

**> {'club', 'diamond', 'spade', 'heart'}**

Однако в списках не должно быть вложенных изменяемых элементов.

**tricky\_list = [{'jocker': 'black'}, {'jocker': 'red'}]**

**sad\_set = set(tricky\_list)**

**print(sad\_set)**

**> Traceback (most recent call last):**

**sad\_set = set(tricky\_list)**

**TypeError: unhashable type: 'dict'**

Создание

Способ №1. Воспользовавшись литералом:

**S = {'1', '2', '3'}**

**print(S)**

**> {'1', '2', '3'}**

**print(type(S))**

**> <class 'set'>**

Способ №2. Применив встроенную функцию set():

Чтобы получить аналогичный результат, необходимо передать итерируемый объект (список, строку или кортеж) в качестве аргумента:

**# объявим список L**

**L = ['1', '2', '3']**

**# и предоставим его в set()**

**S\_2 = set(L)**

**print(S\_2)**

# так как set — коллекция неупорядоченная, то результат вывода может отличаться

> {'1', '2', '3'}

**print(type(S\_2))**

**> <class 'set'>**

Замечание: пустое множество создаётся исключительно через set()

**empty\_set = set()**

**print(empty\_set)**

**> set()**

**print(type(empty\_set))**

**> <class 'set'>**

**Если же сделать так:**

**another\_empty\_set = {}**

**print(another\_empty\_set)**

**> {}**

**print(type(another\_empty\_set))**

**> <class 'dict'>**

То получим пустой словарь. А если внутри фигурных скобок поместить пустую строку:

**maybe\_empty\_set = {''}**

**print(maybe\_empty\_set)**

**> {''}**

**print(type(maybe\_empty\_set))**

**> <class 'set'>**

То на выходе увидим множество, состоящее из одного элемента — этой самой пустой строки.

**# количество элементов множества**

**print(len(maybe\_empty\_set))**

**> 1**

Вполне естественно, что пустое множество, при приведении его к логическому типу, тождественно ложно:

true\_or\_false = set()

print(bool(true\_or\_false))

> False

**Пересечение**

В программировании нередки задачи, в которых требуется найти совпадающие элементы двух коллекций. Классическое решение основано на цикле for, но нас интересует другое — то, что строится на использовании set-ов.

**# пусть есть два списка, и нужно отыскать и вывести их одинаковые компоненты**

**unbreakable\_diamond = ['Jotaro', 'Josuke', 'Koichi']**

**golden\_wind = ['Jotaro', 'Koichi', 'Giorno']**

**# & - здесь оператор пересечения**

**overlap = set(unbreakable\_diamond) & set(golden\_wind)**

**print(overlap)**

**> {'Jotaro', 'Koichi'}**

**Добавление элемента**

Для добавления нового элемента в существующий набор используем метод add(x).

**stats = {1.65, 2.33, 5.0}**

**stats.add(14.7)**

**print(stats)**

**> {1.65, 2.33, 5.0, 14.7}**

Если среди исходных объектов, составляющих set, "x" уже был, то ничего не произойдёт, и начальное множество не изменится.

**big\_cats = {'tiger', 'liger', 'lion', 'cheetah', 'leopard', 'cougar'}**

**big\_cats.add('cheetah')**

**# это жестоко, но второго гепарда не появится**

**print(big\_cats)**

**> {'cheetah', 'liger', 'cougar', 'lion', 'tiger', 'leopard'}**

**Удаление и очистка**

Очистить и свести уже существующий сет к пустому не составит никаких проблем благодаря методу сlear():

**set\_with\_elements = {'i am element', 'me too'}**

**print(set\_with\_elements)**

**> {'i am element', 'me too'}**

**set\_with\_elements.clear()**

**print(set\_with\_elements)**

**> set()**

Для удаления одного единственного компонента из набора в Питоне определены аж три способа.

**Способ №1. Метод remove().**

Метод удаляет элемент elem из set-а. В случае отсутствия elem в наборе интерпретатор выбрасывает исключение.

**point\_coord = {('x', 52.4), ('y', -5), ('z', 0.3)}**

**print(point\_coord)**

**> {('y', -5), ('z', 0.3), ('x', 52.4)}**

**point\_coord.remove(('x', 52.4))**

**print(point\_coord)**

**> {('y', -5), ('z', 0.3)}**

**point\_coord.remove(('z', 11.8))**

**print(point\_coord)**

**> Traceback (most recent call last):**

**point\_coord.remove(('z', 11.8))**

**KeyError: ('z', 11.8)**

**Способ №2. Метод discard().**

Производит предельно схожую с remove() операцию с той лишь разницей, что, в случае отсутствия элемента в коллекции, исключение не возникает:

**triangle\_coord = {(0, 4), (3, 0), (-3, 0)}**

**print(triangle\_coord)**

**> {(3, 0), (-3, 0), (0, 4)}**

**triangle\_coord.discard((0, 4))**

**print(triangle\_coord)**

**> {(3, 0), (-3, 0)}**

**triangle\_coord.discard((54, 55))**

**print(triangle\_coord)**

**> {(3, 0), (-3, 0)}**

Способ №3. Метод pop().

Удаляет и возвращает случайный элемент множества:

**sweets = {'candy', 'gingerbread', 'lollipop', 'candy floss', 'sweets'}**

**print(sweets)**

**> {'lollipop', 'candy', 'sweets', 'gingerbread', 'candy floss'}**

**print(sweets.pop())**

**> lollipop**

**print(sweets)**

**> {'candy', 'sweets', 'gingerbread', 'candy floss'}**

Перебор элементов

Множество, как и любую другую коллекцию, итерируем циклом for:

**iterate\_me = {1.1, 1.2, 1.3, 1.4, 1.5}**

**for num in iterate\_me:**

**print(num)**

**>**

**1.1**

**1.4**

**1.3**

**1.2**

**1.5**

**Принадлежность объекта set-у**

Оператор in даёт возможность проверить наличие элемента в наборе:

**berry\_club = {'Tomato', 'Currant', 'Sea buckthorn', 'Grape', 'Barberry'}**

**print('Tomato' in berry\_club)**

**> True**

**print('Strawberry' in berry\_club)**

**> False**

**22.3 Сортировка множеств**

Операция сортировки отсутствует для множеств Python по определению. **Множество** — неупорядоченный набор. Но не нужно расстраиваться. С помощью функции sorted(), вы всегда можете получить отсортированный список:

**some\_digits = {1, 55, 34, 2, 12, 14, -4}**

**print(sorted(some\_digits))**

**> [-4, 1, 2, 12, 14, 34, 55]**

**cities = {'Москва', 'Калининград', 'Новосибирск', 'Архангельск', 'Белгород', 'Хабаровск'}**

**print(sorted(cities))**

**> ['Архангельск', 'Белгород', 'Калининград', 'Москва', 'Новосибирск', 'Хабаровск']**

Длина множества

Размеры определенного set-а получаем функцией len():

**eng\_alph = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'}**

**print(len(eng\_alph))**

**> 26**

**22.4 Операции на множествах**

**Объединение**

(оператор |, логическое ИЛИ)

Объединением двух множеств "X" и "Y" является такое третье множество "Z", каждый элемент которого принадлежит либо множеству "X", либо "Y".

**lang\_X = {'C++', 'Perl', 'PHP'}**

**lang\_Y = {'Java', 'C#', 'PHP', 'Python'}**

**lang\_Z = lang\_X.union(lang\_Y) # или так lang\_Z = lang\_X | lang\_Y**

**print(lang\_Z)**

**> {'Java', 'C#', 'Python', 'PHP', 'C++', 'Perl'}**

**Пересечение**

(оператор &, логическое И)

Пересечением двух множеств "A" и "B" является такое третье множество "C", каждый элемент которого принадлежит и множеству "A", и множеству "B".

**bats\_enemies = {'Darkside', 'Jocker', 'Bane'}**

**sups\_enemies = {'General Zod', 'Darkside', 'Lobo'}**

**JL\_enemies = bats\_enemies.intersection(sups\_enemies)**

**# или так JL\_enemies = bats\_enemies & sups\_enemies**

**print(JL\_enemies)**

**> {'Darkside'}**

**Разность множеств**

**(оператор -)**

Разностью двух множеств "O" и "P" является такое третье множество "S", каждый элемент которого принадлежит множеству "O" и не принадлежит множеству "P".

**minuend = {23, 44, 1, 34, 98}**

**subtrahend = {23, 44, 1, 55, 76}**

**total = minuend.difference(subtrahend) # или так total = minuend - subtrahend**

**print(total)**

**> {34, 98}**

**Симметрическая разность**

**(оператор ^)**

Симметрической разностью двух множеств "M" и "N" является такое третье множество "L", каждый элемент которого принадлежит либо множеству "M", либо "N", но не их пересечению.

**f\_set = {11, 'a', 18, 'v', 65, 'g'}**

**s\_set = {11, 'z', 32, 'v', 0, 'g'}**

**t\_set = f\_set.symmetric\_difference(s\_set) # или так t\_set = f\_set ^ s\_set**

**print(t\_set)**

**> {32, 0, 65, 18, 'a', 'z'}**

Помимо теоретико-множественных операций, в питоне существуют и сугубо утилитарные производные методы.

**isdisjoint()**

Метод определяет, есть ли у двух set-ов общие элементы:

**it = {'green', 'white', 'red'}**

**ru = {'white', 'blue', 'red'}**

**ukr = {'blue', 'yellow'}**

**# вернет False, если множества пересекаются**

**print(ukr.isdisjoint(it))**

**> True**

**# и True, в противном случае**

**print(ru.isdisjoint(it))**

**> False**

В Python нет оператора, который бы соответствовал этому методу.

**issubset()**

Показывает, является ли "I" подмножеством "J" (Метод вернет True, если все элементы "I" принадлежат "J"):

**solar\_system = {'Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune'}**

**first\_three\_planets = {'Mercury', 'Venus', 'Earth'}**

**poor\_small\_guy = {'Pluto'}**

**emptyness = set()**

**print(first\_three\_planets.issubset(solar\_system))**

**# или так first\_three\_planets <= solar\_system**

**> True**

**print(poor\_small\_guy.issubset(solar\_system))**

**# poor\_small\_guy <= solar\_system**

**> False**

**# как и в математике, пустое множество есть подмножество любого множества**

**print(emptyness.issubset(solar\_system))**

**# emptyness <= solar\_system**

**> True**

**# также любое множество является подмножеством самого себя**

**print(poor\_small\_guy.issubset(poor\_small\_guy))**

**# poor\_small\_guy <= poor\_small\_guy**

**> True**

Оператор < определяет, является ли одно множество строгим подмножеством другого. В большинстве ситуаций, он ведёт себя точно так же, как и <=, кроме последнего случая. Сравниваемые наборы не должны быть идентичными. Таким образом, для данного оператора не существует соответствующего метода.

**print(poor\_small\_guy.issubset(poor\_small\_guy))**

**# poor\_small\_guy < poor\_small\_guy**

**> False**

issuperset()

Показывает, является ли "F" надмножеством "G":

**print(solar\_system.issuperset(first\_three\_planets))**

**# solar\_system >= first\_three\_planets**

**> True**

**print(poor\_small\_guy.issuperset(solar\_system))**

**# poor\_small\_guy >= solar\_system**

**> False**

**# в сердечке Плутона лишь пустота…**

**print(poor\_small\_guy.issuperset(emptyness))**

**# poor\_small\_guy >= emptyness**

**> True**

Особенности оператора строгого надмножества > идентичны таковым у <.

**print(poor\_small\_guy > poor\_small\_guy)**

**> False**

И для него в языке Python тоже не существует соответствующего метода.

Квартет методов, которые присваивают set-у результат его объединения с другим множеством. "Другое множество" передаётся методу в качестве аргумента.

**update()**

Изменяет исходное множество по объединению:

**dogs\_in\_first\_harness = {'Lessie', 'Bork', 'Spark'}**

**dogs\_in\_second\_harness = {'Lucky'}**

**dogs\_in\_second\_harness.update(dogs\_in\_first\_harness)**

**# или так dogs\_in\_second\_harness |= dogs\_in\_first\_harness**

**print(dogs\_in\_second\_harness)**

**> {'Spark', 'Bork', 'Lucky', 'Lessie'}**

**intersection\_update()**

По пересечению:

**basin\_measurements\_west = {-8745, -9000, -7990}**

**basin\_measurements\_east = {-7990, -6934, -8100}**

**basin\_measurements\_west.intersection\_update(basin\_measurements\_east)**

**# или так basin\_measurements\_west &= basin\_measurements\_east**

**print(basin\_measurements\_west)**

**> {-7990}**

**difference\_update()**

По разности:

**prices\_may = {100, 200, 125}**

**prices\_june = {100, 200, 300}**

**prices\_may.difference\_update(prices\_june) # или так prices\_may -= prices\_june**

**print(prices\_may)**

**> {125}**

**symmetric\_difference\_update()**

И, наконец, по симметрической разности:

**his\_bag = {'croissant', 'tea', 'cookies'}**

**her\_bag = {'tea', 'cookies', 'chocolate', 'waffles'}**

**her\_bag.symmetric\_difference\_update(his\_bag)**

**print(her\_bag) # или так her\_bag ^= his\_bag**

**> {'croissant', 'chocolate', 'waffles'}**

**Свойства методов и операторов**

Как показано выше, данные операции, за некоторым исключением, выполнятся двумя способами: при помощи метода или соответствующего ему оператора (например union() и оператор |). Главным и основным их различием является то, что метод может принимать в качестве аргумента не только set, но и любой итерируемый объект, в то время, как оператор требует в качестве операндов наличие фактических множеств.

**list\_of\_years = [2019, 2018, 2017]**

**set\_of\_years = {2009, 2010, 2011}**

**print(set\_of\_years.union(list\_of\_years))**

**> {2017, 2018, 2019, 2009, 2010, 2011}**

**print(set\_of\_years | list\_of\_years)**

**>**

**Traceback (most recent call last):>**

**print(set\_of\_years | list\_of\_years)**

**TypeError: unsupported operand type(s) for |: 'set' and 'list'**

Но есть и сходства. Например, важным является то, что некоторые операторы и методы позволяют совершать операции над несколькими сетами сразу:

**one = set('11111')**

**two = set('22222')**

**three = set('33333')**

**four = set('44444')**

**five = set('55555')**

**uni\_set\_v1 = one.union(two, three, four, five)**

**print(uni\_set\_v1)**

**> {'2', '1', '5', '3', '4'}**

**uni\_set\_v2 = one | two | three | four | five**

**print(uni\_set\_v2)**

**> {'2', '1', '5', '3', '4'}**

**es1 = {11, 21, 31, 311, 3111}**

**es2 = {111, 211, 311, 411}**

**es3 = {1111, 2111, 3111, 4111}**

**# порядок выполнения операций слева --> направо**

**print(es1 - es2 - es3)**

**> {11, 21, 31}**

**print(es1.difference(es2, es3))**

**> {11, 21, 31}**

Тем интереснее, что оператор ^ симметрической разности позволяет использовать несколько наборов, а метод symmetric\_difference() — нет.

**tc1 = {10.1, 20.2, 30.3, 40.4, 50.5}**

**tc2 = {10.1, 20.2, 30.3, 40.4, 500}**

**tc3 = {1, 50.1, 1000}**

**print(tc1 ^ tc2 ^ tc3) # вы же помните про порядок операций (слева-направо)?**

**> {1, 1000, 50.1, 50.5, 500}**

**print(tc1.symmetric\_difference(tc2, tc3))**

**> Traceback (most recent call last):**

**print(tc1.symmetric\_difference(tc2, tc3))**

**TypeError: symmetric\_difference() takes exactly one argument (2 given)**

**22.4 Преобразования**

**Конвертация строки во множество**

Чтобы перевести строку во множество, достаточно представить её в виде литерала этого множества.

**my\_string = 'Lorem ipsum dolor sit amet'**

**sting\_to\_set = {my\_string}**

**print(sting\_to\_set)**

**> {'Lorem ipsum dolor sit amet'}**

**Конвертация списка во множество**

Со списком подобный трюк не пройдет, но здесь на помощь спешит функция set():

**my\_list = [2, 4, 8, 16, 32]**

**list\_to\_set = set(my\_list)**

**print(list\_to\_set)**

**> {32, 2, 4, 8, 16}**

**22.5 Замороженные множества (FrozenSets) в Python**

Замороженные множества являются неизменяемой формой обычных множеств, то есть элементы замороженного множества не могут быть изменены и поэтому могут быть использованы в качестве ключа в словаре.

Элементы замороженного множества не могут быть изменены после создания. Мы не можем изменять или добавлять содержимое замороженных наборов с помощью таких методов, как add() или remove().

Метод frozenset() используется для создания объекта frozenset. В этот метод передается итерируемая последовательность, которая преобразуется в замороженный набор в качестве возвращаемого типа метода.

Для создания замороженного набора рассмотрим следующий пример.

**Frozenset = frozenset([1,2,3,4,5])**

**print(type(Frozenset))**

**print("\nprinting the content of frozen set...")**

**for i in Frozenset:**

**print(i);**

**Frozenset.add(6)**

**Словари**

**23.1 Введение**

В языке программирования Python словари (тип dict) представляют собой еще одну разновидность структур данных наряду со списками и кортежами. Словарь - это изменяемый (как список) неупорядоченный (в отличие от строк, списков и кортежей) набор элементов "ключ:значение".

"Неупорядоченный" – значит, что последовательность расположения пар не важна, в следствие чего обращение к элементам по индексам невозможно.

**23.2 Работа с словарями**

Для создания словаря в Python необходимо передать последовательность элементов внутри фигурных скобок {}, разделив их запятыми (,). Каждый элемент имеет ключ и значение, выраженное парой «ключ: значение».

Значения могут быть представлять собой любые типы данных и повторяться, но ключи обязаны быть уникальными.

Следующие примеры показывают, как создавать словари Python:

Создание пустого словаря:

**dict\_sample = {}**

Cловарь, где ключи являются целыми числами:

**dict\_sample = {1: 'mango', 2: 'pawpaw'}**

Создание словаря с ключами разных типов:

**dict\_sample = {'fruit': 'mango', 1: [4, 6, 8]}**

Можно также создать словарь, явно вызвав метод dict():

**dict\_sample = dict({1:'mango', 2:'pawpaw'})**

Словарь можно создать с помощью последовательности, как в примере внизу:

**dict\_sample = dict([(1,'mango'), (2,'pawpaw')])**

Словари могут быть вложенными. Это значит, что можно создавать словари внутри существующего словаря. Например:

**dict\_sample = {**

**1: {'student1': 'Nicholas', 'student2': 'John', 'student3': 'Mercy'},**

**2: {'course1': 'Computer Science', 'course2': 'Mathematics', 'course3': 'Accounting'}**

**}**

Чтобы вывести содержимое словаря, можно использовать функцию print() и передать название словаря в качестве аргумента. Например:

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**print(dict\_sample)**

**Вывод:**

**{'Company': 'Toyota', 'model': 'Premio', 'year': 2012}**

**Доступ к элементами**

Чтобы получить доступ к элементам словаря, нужно передать ключ в квадратных скобках []. Например:

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**x = dict\_sample["model"]**

**print(x)**

**Вывод:**

**Premio**

Был создан словарь dict\_sample. Затем была создана переменная x. Ее значение — это значение ключа ["model"] из словаря.

Вот другой пример:

**dict = {'Name': 'Mercy', 'Age': 23, 'Course': 'Accounting'}**

**print("Student Name:", dict['Name'])**

**print("Course:", dict['Course'])**

**print("Age:", dict['Age'])**

**Вывод:**

**Student Name: Mercy**

**Course: Accounting**

**Age: 23**

Объект словаря также имеет функцию get(), которой можно пользоваться для доступа к элементам словаря. Ее нужно добавлять к словаря через точку и затем передавать название ключа как аргумент функции. Например:

**ict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**x = dict\_sample.get("model")**

**print(x)**

**Вывод:**

**Premio**

Теперь вы знаете, как получать доступ к элементам словаря с помощью разных методов. В следующем разделе речь пойдет о добавлении новых элементов в уже существующий словарь.

**23.2 Добавление элементов**

Существует множество способов для добавления новых элементов в словарь. Можно использовать новый ключ и присвоить ему значение. Например:

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**dict\_sample["Capacity"] = "1800CC"**

**print(dict\_sample)**

**Вывод:**

**{'Capacity': '1800CC', 'year': 2012, 'Company': 'Toyota', 'model': 'Premio'}**

У нового элемента ключ "Capacity" и значение — "180CC". Он был добавлен в качестве первого элемента словаря.

Вот другой пример. Для начала нужно создать пустой словарь:

**MyDictionary = {}**

**print("An Empty Dictionary: ")**

**print(MyDictionary)**

**Вывод:**

**An Empty Dictionary:**

Словарь ничего не возвращает, потому что в нем ничего не хранится. Добавим в нему элементы, один за одним:

**MyDictionary[0] = 'Apples'**

**MyDictionary[2] = 'Mangoes'**

**MyDictionary[3] = 20**

**print("\n3 elements have been added: ")**

**print(MyDictionary)**

**Вывод:**

**3 elements have been added: {0: 'Apples', 2: 'Mangoes', 3: 20}**

Для добавления элементов были отдельно указаны ключи и соответствующие значения. Например:

**MyDictionary[0] = 'Apples'**

В этом примере 0 является ключом, а "Apples" — значение.

Можно даже добавить несколько значений для одного ключа. Например:

**MyDictionary['Values'] = 1, "Pairs", 4**

**print("\n3 elements have been added: ")**

**print(MyDictionary)**

Вывод:

**3 elements have been added: {'Values': (1, 'Pairs', 4)}**

В этом примере название ключа — "Value", а все что после знака = — его значения в формате множества (Set).

**23.3 Обновление элементов**

После добавления значения в словарь существующий элемент словаря можно изменить. Для изменения значения используется соответствующий ключ. Например:

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**dict\_sample["year"] = 2014**

**print(dict\_sample)**

**Вывод:**

**{'year': 2014, 'model': 'Premio', 'Company': 'Toyota'}**

В этом примере видно, что было обновлено значение ключа "year" с 2012 на 2014.

**Удаление элементов**

Удалить элемент из словаря можно несколькими способами. В этом разделе они будут рассмотрены по одному:

Ключевое слово del можно использовать для удаления элемента с конкретным ключом. Например:

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**del dict\_sample["year"]**

**print(dict\_sample)**

**Вывод:**

**{'Company': 'Toyota', 'model': 'Premio'}**

Вызывается ключевое слово del, а следом за ним — название словаря. В квадратных скобках следом за словарем идет ключ элемента, который требуется удалить. В этом примере это "year". Запись "year" удаляется из словаря.

Другой способ удалить пару ключ-значение — функция pop() с ключом записи в виде аргумента. Например:

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**dict\_sample.pop("year")**

**print(dict\_sample)**

**Вывод:**

**{'Company': 'Toyota', 'model': 'Premio'}**

Функция pop() была вызвана добавлением ее к названию словаря. В этом случае будет удалена запись с ключом "year".

Функция popitem() удаляет последний элемент в словаре. Для нее не нужно указывать конкретный ключ. Примеры:

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**dict\_sample.popitem()**

**print(dict\_sample)**

**Вывод:**

**{'Company': 'Toyota', 'model': 'Premio'}**

Последней записью в словаре была "year". Она пропала из словаря после вызова функции popitem().

Что делать, если нужно удалить целый словарь? Это будет сложно и займет много времени, если пользоваться этими методами к каждому ключу. Вместо этого можно использовать ключевое слово del для целого словаря. Например:

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**del dict\_sample**

**print(dict\_sample)**

**Вывод:**

**NameError: name 'dict\_sample' is not defined**

Код вернет ошибку, потому что функция print() пытается получить доступ к словарю, который уже не существует.

В определенных случаях может потребоваться удалить все элементы словаря, оставив его пустым. Этого можно добиться, воспользовавшись функцией clear():

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**dict\_sample.clear()**

**print(dict\_sample)**

Вывод:

**{}**

Код вернет пустой словарь, поскольку все его элементы уже удалены.

**Другие распространенные методы словарей**

**Метод len()**

**С помощью этого метода можно посчитать количество элементов в словаре. Например:**

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**print(len(dict\_sample))**

Вывод:

**3**

**В этом словаре три записи, поэтому метод вернет 3.**

**Метод copy()**

Этот метод возвращает копию существующего словаря. Например:

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**x = dict\_sample.copy()**

**print(x)**

Вывод:

**{'Company': 'Toyota', 'year': 2012, 'model': 'Premio'}**

Была создана копия словаря dict\_sample. Она присвоена переменной x. Если вывести x в консоль, то в ней будут те же элементы, что и в словаре dict\_sample.

Это удобно, потому что изменения в скопированном словаре не затрагивают оригинальный словарь.

**Метод items()**

Этот метод возвращает итерируемый объект. Такой объект содержит пары ключ-значение для словаря по аналогии с кортежами в списке. Метод используется, когда нужно перебрать значения словаря.

Этот метод нужно вызывать вместе со словарем, как в примере ниже**:**

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**for k, v in dict\_sample.items():**

**print(k, v)**

**Вывод:**

**('Company', 'Toyota')**

**('model', 'Premio')**

**('year', 2012)**

Объект, который возвращает items(), можно использовать, чтобы показать изменения в словаре. Вот как это работает.

**dict\_sample = {**

**"Company": "Toyota",**

**"model": "Premio",**

**"year": 2012**

**}**

**x = dict\_sample.items()**

**print(x)**

**dict\_sample["model"] = "Mark X"**

**print(x)**

**Вывод:**

**dict\_items([('Company', 'Toyota'), ('model', 'Premio'), ('year', 2012)])**

**dict\_items([('Company', 'Toyota'), ('model', 'Mark X'), ('year', 2012)])**

Вывод демонстрирует, что когда вы меняете значение в словаре, объекты элементов также обновляются.

**Метод fromkeys()**

Этот метод возвращает словарь с указанными ключами и значениями. У него следующий синтаксис:

**dictionary.fromkeys(keys, value)**

Значение требуемого параметра keys — итерируемые объекты. Оно отвечает за ключи нового словаря. Значение для параметра value указывать необязательно. Оно отвечает за значение по умолчанию для всех ключей. По умолчанию — None.

Предположим, что нужно создать словарь с тремя ключами и одинаковым значением. Это можно сделать следующим образом:

**name = ('John', 'Nicholas', 'Mercy')**

**age = 25**

**dict\_sample = dict.fromkeys(name, age)**

**print(dict\_sample)**

Вывод:

**{'John': 25, 'Mercy': 25, 'Nicholas': 25}**

В коде вверху определены ключи и одно значение. Метод fromkeys() перебирает ключи и объединяет их со значением для создания заполненного словаря.

Значение для параметра keys является обязательным. В следующем примере показано, что происходит, если параметр values не определен:

**name = ('John', 'Nicholas', 'Mercy')**

**dict\_sample = dict.fromkeys(name)**

**print(dict\_sample)**

Вывод:

**{'John': None, 'Mercy': None, 'Nicholas': None}**

Используется значение по умолчанию, None.