**Тема занятия №10: Итераторы, контейнеры и перечисления**

**1. Общая информация**

Язык Python поддерживает средства для создания классов особого назначения: итераторов, контейнеров и перечислений.

**Итераторы** — это классы, генерирующие последовательности каких-либо значений. Такие классы мы можем задействовать, например, в циклах for:

**class MyIterator: # Определяем класс-итератор**

**. . . . . .**

**it = MyIterator() # Создаем его экземпляр**

**for v in it: # и используем в цикле for**

Контейнеры - классы, которые могут выступать как последовательности (списки или кортежи) или отображения (словари). Мы можем обратиться к любому элементу экземпляра такого класса через его индекс или ключ:

**class MyList: # Определяем класс-список**

**. . . . . .**

**class MyDict: # Определяем класс-словарь**

**. . . . . .**

**lst, dct = MyList(), MyDict() # Используем их**

**lst[0] = 1**

**dct ["first"] = 578**

**Перечисления** - особые классы, представляющие наборы каких-либо именованных величин, в этом смысле они аналогичны подобным типам данных, доступным в других языках программирования, - например, в С:

**from enum import Enum # Импортируем базовый класс Enum**

**class Versions(Enum): # Определяем класс-перечисление**

**Python2.7 = "2.7"**

**Python3.4 = "3.4"**

**# Используем его**

**if python\_version == Versions.Python3.4:**

Поддержка перечислений появилась в Python 3.4.

**2. Итераторы**

**Итераторы** — объекты, которые позволяют обходить коллекции. Коллекции не должны обязательно существовать в памяти и быть конечными. Официальную документацию смотрите по ссылке. Примечание редактора: также смотрите наш курс по спискам.

Давайте уточним определения. Итерируемый — объект, в котором есть метод \_\_iter\_\_. В свою очередь, итератор — объект, в котором есть два метода: \_\_iter\_\_ и \_\_next\_\_. Почти всегда iterator возвращает себя из метода \_\_iter\_\_, так как они выступают итераторами для сами себя, но есть исключения.

В целом стоит избегать прямого вызова \_\_iter\_\_ и \_\_next\_\_. При использовании for или генераторов списков Python вызывает эти методы сам. Если всё-таки необходимо вызвать методы напрямую, используйте встроенные функции iter и next и в параметрах передавайте итератор или контейнер. Например, если c — итерируемый, используйте конструкцию iter(c) вместо c.\_\_iter\_\_(). Если a — итератор, используйте next(a), а не a.\_\_next\_\_(). Это похоже на использование len.

Раз уж речь зашла о len, то стоит упомянуть, что итераторы не должны иметь и часто не имеют определённой длины. Поэтому в них часто нет имплементации \_\_len\_\_. Чтобы подсчитать количество элементов в итераторе, приходится делать это вручную или использовать sum. Пример есть ниже после раздела о модуле itertools.

Некоторые итерируемые (iterable) не являются итераторами, но используют другие объекты как итераторы. Например, объект list относится к итерируемым, но не является итератором. В нём реализован метод \_\_iter\_\_, но отсутствует метод \_\_next\_\_. Итераторы объектов list относятся к типу listiterator. Обратите внимание, у объектов list есть определённая длина, а у listiterator нету.

**>>> a = [1, 2]**

**>>> type(a)**

**<type 'list'>**

**>>> type(iter(a))**

**<type 'listiterator'>**

**>>> it = iter(a)**

**>>> next(it)**

**1**

**>>> next(it)**

**2**

**>>> next(it)**

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

**File "<stdin>", line 1, in <module>**

**StopIteration**

**>>> len(a)**

**2**

**>>> len(it)**

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

**File "<stdin>", line 1, in <module>**

**TypeError: object of type 'listiterator' has no len()**

Когда итератор завершает работу, интерпретатор Python ожидает возбуждения исключения StopIteration. Однако, как уже отмечалось, итераторы могут работать с бесконечными множествами. В таких случаях программист должен позаботиться о выходе из цикла.

Ниже дан простой пример итератора. Он считает с нуля до бесконечности. Это упрощённая версия itertools.count.

**class count\_iterator:**

**n = 0**

**def \_\_iter\_\_(self):**

**return self**

**def \_\_next\_\_(self):**

**y = self.n**

**self.n += 1**

**return y**

Посмотрите пример использования. В последней строке сделана попытка превратить итератор в список. Это приводит к бесконечному циклу.

**>>> counter = count\_iterator()**

**>>> next(counter)**

**0**

**>>> next(counter)**

**1**

**>>> next(counter)**

**2**

**>>> next(counter)**

**3**

**>>> list(counter) # This will result in an infinite loop!**

Важная поправка к сказанному выше: если у объекта нет метода \_\_iter\_\_, его можно обойти, если определить метод \_\_getitem\_\_. В этом случае встроенная функция iter возвращает итератор с типом iterator, который использует \_\_getitem\_\_ для обхода элементов списка. Этот метод возвращает StopIteration или IndexError, когда обход завершается. Пример:

**class SimpleList(object):**

**def \_\_init\_\_(self, \*items):**

**self.items = items**

**def \_\_getitem\_\_(self, i):**

**return self.items[i]**

**И пример использования:**

**>>> a = SimpleList(1, 2, 3)**

**>>> it = iter(a)**

**>>> next(it)**

**1**

**>>> next(it)**

**2**

**>>> next(it)**

**3**

**>>> next(it)**

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

**File "<stdin>", line 1, in <module>**

**StopIteration**

Рассмотрим ещё один интересный пример: генерацию последовательности Q Хофштадтера. В приведённом ниже коде итератор используется для генерации последовательности с помощью вложенных повторений.

Q(n)=Q(n−Q(n−1))+Q(n−Q(n−2))

Например, qsequence([1, 1]) генерирует точную последовательность Хофштадтера. Мы используем исключение StopIteration, чтобы показать, что последовательность не может продолжаться, так как для генерации следующего элемента должен использоваться несуществующий индекс. Если в параметрах укзать значения [1, 2], последовательность немедленно заканчивается.

**class qsequence:**

**def \_\_init\_\_(self, s):**

**self.s = s[:]**

**def \_\_next\_\_(self):**

**try:**

**q = self.s[-self.s[-1]] + self.s[-self.s[-2]]**

**self.s.append(q)**

**return q**

**except IndexError:**

**raise StopIteration()**

**def \_\_iter\_\_(self):**

**return self**

**def current\_state(self):**

**return self.s**

**Вот пример использования:**

**>>> Q = qsequence([1, 1])**

**>>> next(Q)**

**2**

**>>> next(Q)**

**3**

**>>> [next(Q) for \_\_ in range(10)]**

**[3, 4, 5, 5, 6, 6, 6, 8, 8, 8]**

**3. Генераторы**

**Генераторами** называют итераторы, определение которых выглядит как определение функций. Ещё одно определение: генераторы — функции, которые внутри используют выражение yield. Генераторы не могут возвращать значения, вместо этого выдают элементы по готовности. Python автоматизирует запоминание контекста генератора, то есть текущий поток управления, значение локальных переменных и так далее. Каждый вызов метода \_\_next\_\_ у объекта генератора возвращает следующее значение. Метод \_\_iter\_\_ также реализуется автоматически. То есть генераторы можно использовать везде, где требуются итераторы. Пример кода ниже похож на ранее рассмотренный класс итератора. Но этот код более компактный и читабельный.

**def count\_generator():**

**n = 0**

**while True:**

**yield n**

**n += 1**

Посмотрите, как это применяется на практике.

**>>> counter = count\_generator()**

**>>> counter**

**<generator object count\_generator at 0x106bf1aa0>**

**>>> next(counter)**

**0**

**>>> next(counter)**

**1**

**>>> iter(counter)**

**<generator object count\_generator at 0x106bf1aa0>**

**>>> iter(counter) is counter**

**True**

**>>> type(counter)**

**<type 'generator'>**

Теперь взгляните на реализацию последовательности Q Хофштадтера с помощью генератора. Заметьте, эта реализация значительно проще использованного выше подхода. Однако здесь уже невозможно использовать методы типа current\_state. Извне невозможно получить доступ к переменным, которые хранятся в контексте генератора.

Существует словарь gi\_frame.f\_locals, но он относится к CPython, но не входит в стандарт языка Python.

Одно из возможных решений — получение одновременно списка и результата.

**def hofstadter\_generator(s):**

**a = s[:]**

**while True:**

**try:**

**q = a[-a[-1]] + a[-a[-2]]**

**a.append(q)**

**yield q**

**except IndexError:**

**Return**

Обратите внимание, итерация в данном примере завершается простым return без параметров. Внутри происходит возбуждение исключения StopIteration. Следующий пример связан с распределением Бернулли, которое реализуется с помощью двух генераторов. Речь идёт о бесконечной последовательности случайных булевых значений. При этом вероятность True равна p, а вероятность False определяется формулой q=1-p. Затем применяется экстрактор фон Неймана, который принимает процесс Бернулли с 0 < p < 1 как источник энтропии и возвращает чистый процесс Бернулли с p = 0.5.

**import random**

**def bernoulli\_process(p):**

**if p > 1.0 or p < 0.0:**

**raise ValueError("p should be between 0.0 and 1.0.")**

**while True:**

**yield random.random() < p**

**def von\_neumann\_extractor(process):**

**while True:**

**x, y = next(proccess), next(process)**

**if x != y:**

**yield x**

Наконец, с помощью генераторов удобно реализовывать дискретные динамические системы. Пример ниже показывает, как с помощью генераторов реализуется отображение тент.

**>>> def tent\_map(mu, x0):**

**... x = x0**

**... while True:**

**... yield x**

**... x = mu \* min(x, 1.0 - x)**

**...**

**>>>**

**>>> t = tent\_map(2.0, 0.1)**

**>>> for \_\_ in range(30):**

**... print(next(t))**

**...**

**0.1**

**0.2**

**0.4**

**0.8**

**0.4**

**0.8**

**0.4**

**0.8**

**0.4**

**0.8**

**0.4**

**0.8**

**0.4**

**0.8**

**0.4**

**0.8**

**0.4**

**0.799999999999**

**0.400000000001**

**0.800000000003**

**0.399999999994**

**0.799999999988**

**0.400000000023**

**0.800000000047**

**0.399999999907**

**0.799999999814**

**0.400000000373**

**0.800000000745**

**0.39999999851**

**0.79999999702**

Ещё один пример касается последовательности Коллатца.

**def collatz(n):**

**yield n**

**while n != 1:**

**n = n / 2 if n % 2 == 0 else 3 \* n + 1**

**yield n**

Обратите внимание, в этом примере не нужно вручную использовать StopIteration. Это исключение срабатывает автоматически, когда поток управления достигает конца функции.

Пример использования генератора:

**>>> # Если гипотеза Коллатца верна, list(collatz(n)) с любым n**

**... # всегда завершается**

**>>> list(collatz(7))**

**[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]**

**>>> list(collatz(13))**

**[13, 40, 20, 10, 5, 16, 8, 4, 2, 1]**

**>>> list(collatz(17))**

**[17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]**

**>>> list(collatz(19))**

**[19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]**

**4. Рекурсивные генераторы**

Генераторы могут быть рекурсивными, как любая другая функция. Рассмотрим это на примере собственной реализации itertools.permutations. Это генератор перестановок элементов в списке. Обратите внимание, стандартная функция itertools.permutations работает быстрее, поэтому используйте её на практике. Идея простая: функция перемещает каждый элемент списка на первое место, заменяя его первым элементом в списке.

**def permutations(items):**

**if len(items) == 0:**

**yield []**

**else:**

**pi = items[:]**

**for i in range(len(pi)):**

**pi[0], pi[i] = pi[i], pi[0]**

**for p in permutations(pi[1:]):**

**yield [pi[0]] + p**

**>>> for p in permutations([1, 2, 3]):**

**... print(p)**

**...**

**[1, 2, 3]**

**[1, 3, 2]**

**[2, 1, 3]**

**[2, 3, 1]**

**[3, 1, 2]**

**[3, 2, 1]**

**5. Генераторные выражения**

**Генераторные выражения** позволяют реализовать генераторы с помощью очень простого синтаксиса. В примере ниже представлен генератор, который обходит все совершенные квадраты. Результатами генераторных выражений являются объекты с типом generator. В них реализованы методы \_\_next\_\_ и \_\_iter\_\_.

**>>> g = (x \*\* 2 for x in itertools.count(1))**

**>>> g**

**<generator object <genexpr> at 0x1029a5fa0>**

**>>> next(g)**

**1**

**>>> next(g)**

**4**

**>>> iter(g)**

**<generator object <genexpr> at 0x1029a5fa0>**

**>>> iter(g) is g**

**True**

**>>> [next(g) for \_\_ in range(10)]**

**[9, 16, 25, 36, 49, 64, 81, 100, 121, 144]**

Процесс Бернулли из предыдущих разделов также можно реализовать с помощью генераторных выражений. В данном примере p=0.4. Если в генераторном выражении требуется ещё один итератор, который используется в качестве счётчика цикла, можно использовать itertools.count для генерации бесконечной последовательности. В противном случае можно использовать range.

**>>> g = (random.random() < 0.4 for \_\_ in itertools.count())**

**>>> [next(g) for \_\_ in range(10)]**

**[False, False, False, True, True, False, True, False, False, True]**

Как отмечалось выше, генераторные выражения можно передавать в функции, которые нуждаются в итераторе. Например, сумму первых десяти совершенных квадратов можно получить так:

>>> sum(x \*\* 2 for x in xrange(10))

285

**Итераторы, контейнеры и перечисления**

**1. Модуль itertools**

В модуле itertools есть набор итераторов, которые упрощают работу с перестановками, комбинациями, декартовыми произведениями и другими комбинаторными структурами. Документация доступна по ссылке.

Обратите внимание, представленные ниже алгоритмы не являются оптимальными для практического использования. Примеры используются, чтобы показать возможности перестановок и комбинаций. На практике лучше избегать перечисления перестановок и комбинаций, если вы не имеете веской причины для этого, так как размер перечислений растёт по экспоненте.

Посмотрите на интересные примеры ниже. В первом представлен альтернативный способ реализации известного паттерна: обход трёхмерного массива, а также обход всех индексов с 0≤i<j<k≤n.

**from itertools import combinations, product**

**n = 4**

**d = 3**

**def visit(\*indices):**

**print(indices)**

**# Loop through all possible indices of a 3-D array**

**for i in range(n):**

**for j in range(n):**

**for k in range(n):**

**visit(i, j, k)**

**# Equivalent using itertools.product**

**for indices in product(\*([range(n)] \* d)):**

**visit(\*indices)**

**# Now loop through all indices 0 <= i < j < k <= n**

**for i in range(n):**

**for j in range(i + 1, n):**

**for k in range(j + 1, n):**

**visit(i, j, k)**

**# And equivalent using itertools.combinations**

**for indices in combinations(range(n), d):**

**visit(\*indices)**

Альтернативные реализации с использованием перечислителей itertools обеспечивают два преимущества: они короче (одна строка вместо трёх), и их проще обобщать. Нужно помнить, что разница в быстродействии может быть значительной, особенно при больших значениях n.

Второй пример касается интересной математической задачи. С помощью генераторных выражений, itertools.combinations и itertools.permutations вычислим количество инверсий перестановки, а затем суммируем количество инверсий во всех перестановках в списке.

Сумма количества инверсий перестановок n равна OEIS A001809. Поэтому относительно легко показать, что закрытая форма равна n!n(n-1)/4. На практике эта формула эффективнее, но пример кода ниже необходим для отработки применения перечислителей itertools.

**import itertools**

**import math**

**def inversion\_number(a):**

**"""Return the number of inversions in list a."""**

**return sum(1 for x, y in itertools.combinations(range(len(a)), 2) if a[x] > a[y])**

**def total\_inversions(n):**

**"""Return total number of inversions in permutations of n."""**

**return sum(inversion\_number(A) for A in itertools.permutations(range(n)))**

Пример использования:

**>>> [total\_inversions(n) for n in range(10)]**

**[0, 0, 1, 9, 72, 600, 5400, 52920, 564480, 6531840]**

**>>> [math.factorial(n) \* n \* (n - 1) / 4 for n in range(10)]**

**[0, 0, 1, 9, 72, 600, 5400, 52920, 564480, 6531840]**

В качестве ещё одного примера давайте рассчитаем количество повторений с помощью метода полного перебора. Даны n и k, количество повторений Dn,k определяется как количество перестановок множества n, в котором k фиксированных значений.

Сначала пишем функцию, которая использует генераторное выражение для подсчёта фиксированных значений в перестановке. Затем используем itertools.permutations и ещё одно генераторное выражение, чтобы посчитать общее количество перестановок n, в которых k фиксированных значений. Ещё раз обратите внимание, код ниже неоптимальный, он служит для отработки применения генераторных выражений.

**def count\_fixed\_points(p):**

**"""Return the number of fixed points of p as a permutation."""**

**return sum(1 for x in p if p[x] == x)**

**def count\_partial\_derangements(n, k):**

**"""Returns the number of permutations of n with k fixed points."""**

**return sum(1 for p in itertools.permutations(range(n)) if count\_fixed\_points(p) == k)**

Применение:

**# Usage:**

**>>> [count\_partial\_derangements(6, i) for i in range(7)]**

**[265, 264, 135, 40, 15, 0, 1]**

**2. Контейнер**

**Контейнер** — тип, экземпляр которого несёт в себе один или несколько других элементов (объектов).

Контейнеры обычно представляют из себя последовательности (например списки, кортежи) или отображения (словари), но могут являть собой и нечто другое.

Некоторые контейнеры могут предоставлять возможность поочерёдного прохода по своим элементам. О таких последовательностях говорят, что они поддерживают итерирование.

Проверка наличия элемента

Наличие элемента в контейнера проверятся при помощи оператора in:

a = [1, 2, 3]

2 in a # True

5 in a # False

**3. Перечисления**

**Перечисления в Python** — это набор символических имен (членов), привязанных к уникальным постоянным значениям. Внутри перечисления члены могут сравниваться по идентичности, а само перечисление может повторяться.

Так как перечисления используются для представления констант, то рекомендуется использовать стиль написания имен для членов перечисления - UPPER\_CASE, так же этот стиль будет использоваться в примерах.

Модуль enum определяет четыре класса перечисления, которые можно использовать для определения уникальных наборов имен и значений: enum.Enum, enum.IntEnum, enum.Flag и enum.IntFlag. Он также определяет один декоратор @unique() и один помощник enum.auto.

enum.Enum:

Базовый класс enum.Enum предназначен для создания нумерованных констант. Подробнее смотрите в разделе "Альтернативный синтаксис enum.Enum".

enum.IntEnum:

Базовый класс enum.IntEnum предназначен для создания перечисляемых констант, которые также являются подклассами int.

enum.IntFlag:

Базовый класс enum.IntFlag предназначен для создания нумерованных констант, которые можно комбинировать с помощью побитовых операторов без потери членства в enum.IntFlag. Члены базового класса enum.IntFlag также являются подклассами int.

enum.Flag:

Базовый класс enum.Flag предназначен для создания пронумерованных констант, которые можно комбинировать с помощью побитовых операций без потери их принадлежности к базовому классу enum.Flag.

enum.unique():

Декоратор enum.unique() базового класса enum.Enum, обеспечивает привязку только одного имени к любому значению.

enum.auto:

Класс enum.auto заменяет экземпляры членов enum.Enum соответствующим значением. По умолчанию начальное значение начинается с 1.

Новое в Python 3.6: добавлены enum.Flag, enum.IntFlag, enum.auto

Пример использования модуля enum:

>>> from enum import Enum

>>> class Planet(Enum):

... MERCURY = (3.303e+23, 2.4397e6)

... VENUS = (4.869e+24, 6.0518e6)

... EARTH = (5.976e+24, 6.37814e6)

... MARS = (6.421e+23, 3.3972e6)

... JUPITER = (1.9e+27, 7.1492e7)

... SATURN = (5.688e+26, 6.0268e7)

... URANUS = (8.686e+25, 2.5559e7)

... NEPTUNE = (1.024e+26, 2.4746e7)

... def \_\_init\_\_(self, mass, radius):

... self.mass = mass # in kilograms

... self.radius = radius # in meters

... @property

... def surface\_gravity(self):

... # universal gravitational constant (m3 kg-1 s-2)

... G = 6.67300E-11

... return G \* self.mass / (self.radius \* self.radius)

...

>>> Planet.EARTH.value

(5.976e+24, 6378140.0)

>>> Planet.EARTH.surface\_gravity

9.802652743337129