Что нового в Python 3.8

Добавили метод `as_integer_ratio()` для `int` и `bool`

Данный метод был реализован для типа `float` и `decimal`. он возвращал пару интов, числитель и знаменатель данного дробного числа. Теперь работает и для `int` и `bool`. Ниже представлен демонстрация работы данного метода.

```
import decimal
test_int=int(5)
test_float=3.2
test_decimal = decimal.Decimal(3.2)
test_bool = True
print(test_int.as_integer_ratio(),test_bool.as_integer_ratio(),
test_float.as_integer_ratio(), test_decimal.as_integer_ratio())
```

```
(5, 1) (1, 1) (3602879701896397, 1125899906842624) (3602879701896397,
1125899906842624)
```

Модуль Collections

Метод _asdict() для collections.namedtuple() теперь возвращает dict, а не collections.OrderedDict. Это работает потому, что обычные словари имеют гарантированное упорядочивание, начиная с Python 3.7. Ниже приведен пример использования данного метода.

```
import collections
test_namedtuple = collections.namedtuple('Coord',['x','y','z'])
test_coord = test_namedtuple(10,10.5,13)
print(test_coord)
b=test_coord._asdict()
print(b.popitem())
```

```
Coord(x=10, y=10.5, z=13)
('z', 13)
```

Как видно для обычного словаря b работает метод popitem, который выбирает последний элемент, что значит, что в словаре есть порядок.

Обновления в Python 3.9

Dict

Слияние словарей с помощью |

```
test_dict = {'x':1,'y':2}
second_test_dist = {'a':(1,2), 'b':(2,3),'x':(5,7,6)}
print(test_dict | second_test_dist)

{'x': (5, 7, 6), 'y': 2, 'a': (1, 2), 'b': (2, 3)}
```

String

Новые методы удаления префиксов и суффиксов из строк. `str.removeprefix(prefix)` и `str.removesuffix(suffix)`. Также были добавлены соответствующие методы bytes, bytearray и collections.UserString

```
test_str = 'abcdefmakoveev'
test_remove_pr = test_str.removeprefix('abc')
test_remove_suff = test_str.removesuffix('veev')
test_remove_pr2 = test_str.removeprefix('ef')
print(test_remove_pr, test_remove_suff,test_remove_pr2)

defmakoveev abcdefmako abcdefmakoveev
```

Обновления в Python 3.10

Новый тип Union

Появился новый оператор объединения типов, который позволяет использовать синтаксис X | Y. Это обеспечивает более чистый способ выражения "либо тип X, либо тип Y" вместо использования typing.Union, особенно в подсказках типов.

```
from typing import Union


def square(number: Union[int, float]) -> Union[int, float]:
    return number ** 2

test_int_square = square(5)
test_float_square = square(5.5)
print(isinstance(test_int_square, float))
print(isinstance(test_int_square, int | float))
print(isinstance(test_float_square, int))
print(isinstance(test_float_square, int | float))

False
True
False
True
```

Int

В типе int появился новый метод int.bit_count(), возвращающий количество единиц в двоичном расширении данного целого числа, также известный как счетчик населенности.

```
test_int = 2024
print(bin(test_int))
print(test_int.bit_count())
```

```
0b11111101000
7
```

Array

Метод index() в array.array теперь имеет необязательные параметры start и stop.

```
import array
test_array = array.array('b',[1,2,3,4,5])
print(test_array.index(3,1,-1))
try:
    print(test_array.index(1,start=1,stop=-1))
except Exception as e:
    print(e)
```

```
2
array.index() takes no keyword arguments
```

Обновления в Python 3.11

Self type

Новая аннотация Self обеспечивает простой и интуитивно понятный способ аннотирования методов, возвращающих экземпляр своего класса. Этот способ аналогичен подходу на основе TypeVar, описанному в PEP 484, но является более лаконичным и простым.

К распространенным случаям использования относятся альтернативные конструкторы, предоставляемые как методы класса, и методы **enter**(), возвращающие self:

```
from typing import Self


class MyLock:
    def __enter__(self) -> Self:
        self.lock()
        return self
```

```
class MyInt:
    @classmethod
    def fromhex(cls, s: str) -> Self:
        return cls(int(s, 16))
```

String

Добавьте в string.Template функции get_identifiers() и is_valid(), которые, соответственно, возвращают все допустимые заполнители и наличие недопустимых заполнителей.

```
import string

test_string = string.Template("$who best player of $game")
print(test_string.get_identifiers(), test_string.is_valid())
```

```
['who', 'game'] True
```

Теперь рассмотрим типы данных модуля Collections

Все представленные ниже типы данных не являются встроенными. Для их использования нужно импортировать модуль collections(`import collections`)

UserDict, UserList, UserString

Все три типа данных являются оберткой над встроенными классами `dict,list,string`. Нужны они, когда мы хотим создать свой кастомный класс на основе базового типа данных. Изначально в python 2.2 нельзя было так сделать, поэтому и существовали такие типы данных. Но в python 3 прямая необходимость в них отпала, единственное это немного удобнее использовать эти типы даных, тк не надо переживать об переопределение методов. Особо примеров придумать не смог, поэтому вот банальный:

```
from collections import UserDict, UserList


class MyDict(UserDict): # словарь содержащий инты и превращающий их в
hex представление
    def __setitem__(self, key, item: int) -> None:
        item=hex(item)
        return super().__setitem__(key, item)

a = MyDict({'a':125,'b':245,'c':348})
print(a)
```

```
{'a': '0x7d', 'b': '0xf5', 'c': '0x15c'}
```

Deque

Deques - это обобщение стеков и очередей (является сокращением от "двусторонняя очередь"). Deques поддерживают потокобезопасные, экономящие память добавления и удаления с любой стороны deque с примерно одинаковой производительностью O(1) в любом направлении. Основные методы:

1. append(x): Добавляет x в правую часть deque.
2. appendleft(x): Добавляет x в левую часть deque.
3. clear(): Удаляет все элементы из deque, оставив его длину 0.
4. count(x): Подсчитывает количество элементов deque, равных x.
5. extend(iterable) Расширяет правую часть deque, добавляя элементы из аргумента iterable.
6. extendleft(iterable) Расширяет левую часть deque, добавляя элементы из iterable. Обратите внимание, что серия добавлений слева приводит к изменению порядка элементов в аргументе iterable.

С полным списком методов можно ознакомиться на сайте оф. документации https://docs.python.org/3/library/collections.html#collections.deque

В пример будет преведена наивная программа очереди исполняемых процессов. Если процесс требует времени меньше чем 30с, то он ставится в начало очереди. Если больше, то в конец.

```
import random
import collections

possible_name = ['Hair Rush ', 'Fat 2 Fit ', 'Sandman Run ', 'Butt
Clash ', 'Shape-shifting ', 'Clothes Run ', 'Long Neck Run ', 'Juice
Run ', 'Pancake Run ', 'Layers Roll ']
class Process():
    def __init__(self, time=0, name='defaut'):
        self.time=time
        self.name=name+str(self.time)+'s'
myqueue=collections.deque()
for i in range(100):

process=Process(random.randint(0,100),random.choice(possible_name))
    if process.time>=15:
        myqueue.append(process.name)
    else:
        myqueue.appendleft(process.name)
print(myqueue)
```

```
deque(['Juice Run 2s', 'Layers Roll 3s', 'Shape-shifting 9s', 'Clothes
Run 1s', 'Pancake Run 14s', 'Butt Clash 12s', 'Pancake Run 12s',
'Clothes Run 0s', 'Layers Roll 3s', 'Butt Clash 8s', 'Butt Clash 7s',
```

```
'Layers Roll 12s', 'Sandman Run 14s', 'Long Neck Run 5s', 'Butt Clash
12s', 'Sandman Run 12s', 'Sandman Run 2s', 'Clothes Run 55s', 'Sandman
Run 63s', 'Hair Rush 76s', 'Sandman Run 45s', 'Butt Clash 84s',
'Pancake Run 38s', 'Layers Roll 57s', 'Long Neck Run 34s', 'Long Neck
Run 93s', 'Fat 2 Fit 79s', 'Hair Rush 36s', 'Butt Clash 36s', 'Hair
Rush 27s', 'Long Neck Run 42s', 'Clothes Run 65s', 'Butt Clash 85s',
'Shape-shifting 57s', 'Clothes Run 100s', 'Long Neck Run 27s', 'Layers
Roll 39s', 'Long Neck Run 98s', 'Clothes Run 17s', 'Sandman Run 80s',
'Hair Rush 91s', 'Butt Clash 43s', 'Sandman Run 68s', 'Long Neck Run
87s', 'Layers Roll 53s', 'Long Neck Run 15s', 'Shape-shifting 52s',
'Fat 2 Fit 29s', 'Clothes Run 88s', 'Butt Clash 74s', 'Butt Clash
36s', 'Long Neck Run 17s', 'Long Neck Run 47s', 'Shape-shifting 78s',
'Fat 2 Fit 24s', 'Sandman Run 37s', 'Long Neck Run 88s', 'Clothes Run
70s', 'Butt Clash 71s', 'Juice Run 67s', 'Layers Roll 47s', 'Juice Run
67s', 'Shape-shifting 52s', 'Juice Run 73s', 'Shape-shifting 88s',
'Layers Roll 29s', 'Shape-shifting 92s', 'Juice Run 66s', 'Clothes Run
84s', 'Fat 2 Fit 47s', 'Sandman Run 94s', 'Juice Run 97s', 'Fat 2 Fit
39s', 'Layers Roll 57s', 'Pancake Run 76s', 'Fat 2 Fit 57s', 'Juice
Run 74s', 'Butt Clash 65s', 'Juice Run 95s', 'Clothes Run 42s',
'Clothes Run 77s', 'Fat 2 Fit 48s', 'Pancake Run 57s', 'Sandman Run
55s', 'Fat 2 Fit 21s', 'Clothes Run 22s', 'Fat 2 Fit 60s', 'Shape-
shifting 20s', 'Hair Rush 93s', 'Layers Roll 27s', 'Sandman Run 27s',
'Sandman Run 73s', 'Hair Rush 60s', 'Shape-shifting 89s', 'Hair Rush
99s', 'Hair Rush 50s', 'Clothes Run 64s', 'Shape-shifting 61s', 'Butt
Clash 82s', 'Clothes Run 57s'])
```

ChainMap

Это контейнер словарей. Данный тип данных создается от нескольких словарей и содержит их не составляя один словарь.

Поиск последовательно перебирает все базовые отображения, пока не будет найден ключ. В отличие от этого, записи, обновления и удаления работают только с первым отображением.

ChainMap включает в себя базовые отображения по ссылке. Поэтому, если одно из базовых отображений будет обновлено, эти изменения будут отражены в ChainMap.

Поддерживаются все обычные методы работы со словарями. Кроме того, есть атрибут maps, метод для создания новых подконтекстов и свойство для доступа ко всем отображениям, кроме первого.

```python
from collections import ChainMap

# Создаем несколько словарей
dict1 = {'apple': 5, 'banana': 7}
dict2 = {'banana': 10, 'orange': 3}
dict3 = {'orange': 6, 'kiwi': 4}
```

```python
# Создаем ChainMap из этих словарей
chain = ChainMap(dict1, dict2, dict3)

# Выводим значение для 'banana', которое берется из первого словаря, где это значение присутствует
print(chain['banana'])

# Методы класса ChainMap:

# .maps - возвращает список словарей, образующих ChainMap
print(chain.maps)

# .new_child() - создает новый ChainMap с добавленным словарем
dict4 = {'grape': 9, 'lemon': 2}
new_chain = chain.new_child(dict4)
print(new_chain)

# .parents - возвращает новый ChainMap, содержащий все словари, кроме первого
print(chain.parents)

# .get(key, default=None) - получить значение для ключа, если его нет, вернуть default
print(chain.get('kiwi', 'Key not found'))

# .keys() - возвращает ключи всех словарей в ChainMap
print(chain.keys())

# .values() - возвращает значения всех словарей в ChainMap
print(chain.values())

# .items() - возвращает ключи и значения всех словарей в ChainMap в виде кортежей
print(chain.items())
```

```
7
[{'apple': 5, 'banana': 7}, {'banana': 10, 'orange': 3}, {'orange': 6, 'kiwi': 4}]
ChainMap({'grape': 9, 'lemon': 2}, {'apple': 5, 'banana': 7}, {'banana': 10, 'orange': 3}, {'orange': 6, 'kiwi': 4})
ChainMap({'banana': 10, 'orange': 3}, {'orange': 6, 'kiwi': 4})
4
KeysView(ChainMap({'apple': 5, 'banana': 7}, {'banana': 10, 'orange': 3}, {'orange': 6, 'kiwi': 4}))
ValuesView(ChainMap({'apple': 5, 'banana': 7}, {'banana': 10, 'orange': 3}, {'orange': 6, 'kiwi': 4}))
ItemsView(ChainMap({'apple': 5, 'banana': 7}, {'banana': 10, 'orange': 3}, {'orange': 6, 'kiwi': 4}))
```

На этом все)