Глава 3. Словари и множества

Все словари наследуют класс **collections.abc.Mapping**. Ключи должны быть хэшируемые. Включать метод *hash*() и *eq*() Объект называется хэшируемым, если он обладает хэш-значением, которое не изменяется на протяжении всей жизни объекта и допускает сравнение с другими объектами.

Способы инициализации словаря

```
a = dict(one=1, two=2, three=3)
b = {'one': 1, 'two': 2, 'three': 3}
c = dict(zip(['one','two','three'],[1,2,3]))
d = dict([('two', 2), ('one', 1), ('three', 3)])
e = dict({'two': 2, 'one': 1, 'three': 3})
a == b == c == d == e
True
```

Глава 4. Тексты и байты.

Всё что нужно знать о байтах: главное это то, что существует 2 основных типа отображения двоичных последовательностей: изменяемы тип bytes, появившийся в py3 и не изменяемы тип bytearray. Каждый элемент bytes или bytearray - целое число от 0 до 255

Глава 5. Полноправные функции

Семь видов вызываемых объектов

Оператор () можно применять не только к функциям, определённым пользователями. Что бы понять является ли объект вызываемым, воспользуейтесь функцией:

```
callable()
```

Table 1. Вызываемые элементы Python

Функция	Описание
Пользовательские функции	Создаются при помощи выражения `def` или `lambda`-выражения
Встроенные функции	Функции написанные на C (в случае CPython), например `len` или `time.strftime`
Методы	Функции определённые в теле класса
Встроенные методы	Метода написанные на C, например dict.get
Классы	При вызове класса выполняется свой метод `new`, что бы создать экземпляр, затем вызывает метод `init` для его инициализации и, наконец, возвращает экземпляр вызывающей программе
Экземпляры классов	Если в классе определен метод `call`, то его экземпляры можно вызвать, как функции
Генераторные функции	Функции или методы, в которых используется ключевое слово `yield`. При вызове генераторная функция возвращает объект-генератор

Учитывая разнообразие вызываемых типов в Python, самый безопасный способ узнать, является ли объект вызываемым, - воспользоваться встроенной функцией callable()

Пользовательские вызываемые типы

Пример создания класса с реализованным методом `__call__`

```
import random


class BingoCage:
    """
    Экземпляр этого класса строится из любого иттерируемого объекта и
    хранит внутри себя список элементов в случайном порядке. При вызове
    экземпляра из списка удаляется один элемент.
    """

    def __init__(self, items=None):
        """
        Метод __init__ принимает произвольный иттерируемый объект;
        Создание локальной копии предотвращает изменение списка, переданного
        в качестве аргумента.
        """
        self._items = list(items)
        random.shuffle(self._items)  # Метод shuffle гарантированно работает, т.к.
self._items объект тип list.

    def pick(self):
        """
        Основной метод.
        """
        try:
            return self._items.pop()
        except IndexError:
            # Возбудить исключение со специальным сообщением, если список self._items
пустой.
            raise LookupError('pick from empty BingoCage')

    def __call__(self):
        """
        Позволяет писать просто bingo() вместо bingo.pick()
        :return:
        """
        return self.pick()
```

Демонстрация BingoCage

```
>>> from source.bingocall import BingoCage
>>> bingo = BingoCage(range(3))
>>> bingo.pick()
2
>>> bingo()
1
>>> callable(bingo)
True
```

Объект `bingo` можно вызвать как функцию, и встроенная функция `callable(…)` распознает его как вызываемый объект

Пример классного разбора именованых и не именовых аргументов функции

```
def tag(name, *content, cls=None, **attrs):
    """
    Функция tag генерирует HTML; чисто именованый аргумент cls
    для передачи аттрибута "class". Это обходное решение необходимо,
    т.к. в Python class - Зарезервированное слово.
    """
    print(name)
    if cls is not None:
        attrs['class'] = cls
    if attrs:
        attr_str = ''.join(' %s="%s"' % (attr, value) for attr, value in sorted(attrs
.items()))
    else:
        attr_str = ''
    if content:
        return '\n'.join('<%s%s>%s</%s>' % (name, attr_str, c, name) for c in content)
    else:
        return '<%s%s />' % (name, attr_str)
```

Получение информации о параметрах

- У объекта-функции есть аттрибут `defaults`, в котором хранится кортеж со значениями по умолчанию позиционных и именованных параметров.
- Значения чисто именованных аргументов находятся в `kwdefaults`
- Сами имена параметров находятся в атрибуте `code`, который содержится ссылку на объект `code` с множеством своих собственных параметров

```
>>> from source.tag import tag
>>> tag.__code__.co_varnames
('name', 'cls', 'content', 'attrs')
>>> tag.__code__.co_argcount
1
```

inspect.signature

Метод `inspect.signature` возвращает объект inspect.Signature, у которого есть аттрибут `parameters`, позволяющий прочитать упорядоченное отображение имен на объекы типа `inspect.Parameter`. У каждого объекта `Parameter` есть набор аттрибутов, например: `name`, `default` и `kind`. Специально значение `inspect._empty` обозначающий параметры, не имеющие значения по-умолчанию.

```
>>> from inspect import signature
>>> sig = signature(help)
>>> sig
<Signature (*args, **kwds)>
>>> str(sig)
'(*args, **kwds)'
>>> for name, param in sig.parameters.items(): print(param.kind, ':', name, '=',
param.default)
...
VAR_POSITIONAL : args = <class 'inspect._empty'>
VAR_KEYWORD : kwds = <class 'inspect._empty'>
>>> sig = signature(open)
>>> for name, param in sig.parameters.items(): print(param.kind, ':', name, '=',
param.default)
...
POSITIONAL_OR_KEYWORD : file = <class 'inspect._empty'>
POSITIONAL_OR_KEYWORD : mode = r
POSITIONAL_OR_KEYWORD : buffering = -1
POSITIONAL_OR_KEYWORD : encoding = None
POSITIONAL_OR_KEYWORD : errors = None
POSITIONAL_OR_KEYWORD : newline = None
POSITIONAL_OR_KEYWORD : closefd = True
POSITIONAL_OR_KEYWORD : opener = None
```

У объекта `inspect.Signature` имеется метод `bind`, который принимает любое количество атрибутов и связывает их с параметрами, указанных в сигнатуре, следуя обычным правилам сопоставления фактических аргументов с формальными параметрами.

Каркас может использовать эту возможность для проверки аттрибутов до фактического вызова функции.

Связывание сигнатуры функции `tag` со словарём аргументов

```
>>> import inspect
>>> from source.tag import tag
>>> sig = inspect.signature(tag)
>>> my_tag = {
... 'name'  : 'img',
... 'title' : 'Sunset Boulevard',
... 'src'   : 'sunset.jpg',
... 'cls'   : 'framed'
... }
>>> bounds_args = sig.bind(**my_tag)
>>> bounds_args
<BoundArguments (name='img', cls='framed', attrs={'title': 'Sunset Boulevard', 'src':
'sunset.jpg'})>
>>> for name, value in bounds_args.arguments.items(): print(name, '=', value)
...
name = img
cls = framed
attrs = {'title': 'Sunset Boulevard', 'src': 'sunset.jpg'}
>>> del my_tag['name']
>>> bounds_args = sig.bind(**my_tag)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    bounds_args = sig.bind(**my_tag)
  File "C:\Users\User\AppData\Local\Programs\Python\Python38-32\lib\inspect.py", line
3025, in bind
    return self._bind(args, kwargs)
  File "C:\Users\User\AppData\Local\Programs\Python\Python38-32\lib\inspect.py", line
2940, in _bind
    raise TypeError(msg) from None
TypeError: missing a required argument: 'name'
>>>
```

На этом примере видно, как модель данных Python - посредством модуля inspect - раскрывает механизм, которым пользуется сам интерпретатор дя связывания аргументов с формальными параметрами при вызове функии.

Аннотация функций

Аннотированная функция clip

```
def clip(text: str, max_len: 'int > 0' = 80) -> str:  # Аннотированное объявление
функции
    """
    Return text clipped at the last space before or after max_len

    :param text:
        Переменная с текстом
    :param max_len:
        Максимальная длина возвращаемой строки
    :return:
        Возращает строку обрезаную до последнего пробела или до максимальной длины.
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None:  # No spaces were found
        end = len(text)
    return text[:end].rstrip()
```

- У любого аргумента в объявлении функции может быть выражение аннотации, которому предшествует `:`.
- Если у аргумента есть значение по-умолчанию, то аннотация располагается между именем и знаком `=`.
- Что-бы аннотировать возвращаемое значение, поместите `->` и вслед за ним выражение между знаком `)` и двоеточием в конце объявления функции.
- Аннотации никак не обрабатываются. Они просто сохраняются в аттрибуте функции `__annotations__` тип `dict`

```
>>> from source.clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': 'int > 0', 'return': <class 'str'>}
```

Пакеты для функционального программирования

Модуль `operator`

Модуль `operator` включает в себя функции для выборки элементов из последовательностей и чтения атрибутов объектов: `itemgetter` и `attrgetter` строят специализированные функции

для выполнения этих действий.

Результат применения `itemgetter` для сортировки списка кортежей

```
from operator import itemgetter

metro_data = [
    ('Tokyo', 'JP', 36.933, (35, 139)),
    ('Delhi NCR', 'IN', 21.935, (28, 77)),
    ('Mexico City', 'MX', 20.142, (19, -99)),
    ('New York-Newark', 'US', 20.104, (40, -74)),
    ('Sao Paulo', 'BR', 19.649, (-23, -46)),
]

for city in sorted(metro_data, key=itemgetter(1)):
    print(city)
```

```
py .\metro_data.py
('Sao Paulo', 'BR', 19.649, (-23, -46))
('Delhi NCR', 'IN', 21.935, (28, 77))
('Tokyo', 'JP', 36.933, (35, 139))
('Mexico City', 'MX', 20.142, (19, -99))
('New York-Newark', 'US', 20.104, (40, -74))
```

Фиксация аргументов с помощью `functools.partial`

В модуле `functools` собраны некоторые функции высшего порядка. Из них наиболее широко известна функция `reduce`. Помимо неё, особенно полезна функция `partial` и её вариация `partialmethod`.

- `functools.partial` — функция высшего порядка. Позволяет применять функцию "частично". Получив на вход некоторую функцию, `partial` создает новый вызываемый объект, в котором некоторые аргументы исходной функции фиксированы. Функция `partial` принимает в первом аргументе вызываемый объект, а за ним - произвольное число позиционных и именованных аргументов, подлежащих связыванию.

Построение вспомогательной функции нормализации Unicode-строк с помощью partial

```
>>> import functools
>>> import unicodedata
>>> nfc = functools.partial(unicodedata.normalize, 'NFC')
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> nfc(s1) == nfc(s2)
True
>>> s1 == s2
False
```

- `functools.partialmethod` — делает тоже самое, что и `partial`, но предназначена для работы с методами.

Спецификации и статьи по пройденному материалу:

1. PEP 3102 — Keyword-Only Arguments
2. PEP 3107 — Function Annotations
3. PEP 362 — Function Signature Object
4. Functional Programming HOWTO