MACHINE LEARNING

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ PYTHON И РАЗРАБОТКА ПРОГРАММ ДЛЯ МАШИННОГО ОБУЧЕНИЯ

ЛЕКЦИЯ IV

План занятия

- Списки
- Кортежи
- Словари и ключи
- Функции
- Локальные и глобальные переменные
- Рекурсия
- Двумерные массивы
- Вложенные списки и массивы

Списки

Список состоит из элементов, разделенных запятыми, находящихся между квадратными скобками ([ ]). В определенной мере, списки подобны массивам в С. Единственной разницей является то, что элементы одного списка могут иметь разные типы данных.

Получить доступ к элементам, сохраненным в списке можно, точно так же, как и в строках, при помощи оператора нарезки ([ ] и [:]) и индексов, начиная с нуля и до конца. Знак плюс (+) объединяет два списка, а звездочка (*) - оператор повторения для списка.

Пример

```
1  my_list =[True, 786, 3.14, 'text', 70.2]
2  second_list =[123, 'text']
3
4  print my_list           # Напечатает весь список
5  print my_list[0]        # Напечатает первый элемент списка
6  print my_list[1:3]      # Напечатает элементы списка со второго по третий
7  print my_list[2:]       # Напечатает элементы списка начиная с третьего
8  print second_list *2  # Напечатает удвоенный список
9  print my_list + second_list # Напечатает объединенные списки
```

Результат

```
1  [True, 786, 3.14, 'text', 70.2]
2  True
3  [786, 3.14]
4  [3.14, 'text', 70.2]
5  [123, 'text', 123, 'text']
6  [True, 786, 3.14, 'text', 70.2, 123, 'text']
```

Кортежи

Кортеж состоит из ряда значений, разделенных запятыми, заключенными в круглые скобки (( )). Основным различием между списками и кортежами является то, что элементы кортежей не могут быть изменены. То есть, кортежи можно рассматривать как списки доступные только для чтения.

Если у вас нет необходимости изменять элементы списка, то для экономии места в памяти лучше использовать тип данных кортеж.

Пример

```
1  my_tuple =(True, 786, 3.14, 'text', 70.2)
2  second_tuple =(123, 'text')
3
4  print my_tuple # Печатает весь кортеж
5  rint my_tuple[0] # Печатает первый элемент
6  print second_tuple *2 # Печатает удвоенный кортеж
7  print my_tuple + second_tuple # Печатает объединенные кортежи
```

Результат

```
1  (True, 786, 3.14, 'text', 70.2)
2  True
3  (123, 'text', 123, 'text')
4  (True, 786, 3.14, 'text', 70.2, 123, 'text')
```

```
1  my_list = ["Rome", 23, ["cat","dog"], True, 3.14]
2  my_tuple = ("Rome", 23, ["cat","dog"], True, 3.14)
3
4  my_list[0] = "Paris" # Замена значения первого элемента сработает для списка
5  my_tuple[0] = "Paris" # Та же операция для кортежа вызовет ошибку
```

Словари и ключи

Словари в Python это неотсортированная коллекция элементов, доступ к которым осуществляется по ключу. То есть, каждому ключу словаря соответствует определенное значение. Ключом может быть любой неизменяемый тип данных (число, строка, кортеж), значением - любой тип данных.

Пары ключ, значение словаря заключаются в фигурные скобки ({ }).

Есть несколько способов создания словарей:

```
1  my_dict = { } # Создаем пустой словарь
2  my_dict["country"] = "Mexico" # Присваиваем ключу country значение Mexico
3  print my_dict["country"] # Выведет Mexico
4
5  # Заполнение словаря при инициализации
6  another_dict = {"number":23, 2: True, "my_list":[1,2,3]}
7  print another_dict.keys() # Напечатает список всех ключей
8  print another_dict.values() # Напечатает список всех значений
```

```
>>> my_dict = { }
>>> my_dict["country"] = "Mexico"
>>> print my_dict["country"]
Mexico
>>> another_dict = {"number":23, 2:True, "my_list":[1,2,3]}
>>> print another_dict.keys()
[2, 'my_list', 'number']
>>> print another_dict.values()
[True, [1, 2, 3], 23]
>>>
```

Сеты

Сет в Python это еще один изменяемый, коллекционный тип данных, отличительной чертой которого является то, что он хранит только уникальные значения.

Создать сеты можно следующими способами:

```
1  # Создание пустого сета
2  s = set()
3  # Создание сета инициализацией
4  s = {"hi", "bye"}
```

Для добавление элемента в сет используется метод add, для удаления - pop или remove. Добавление в сет уже существующего элемента не повлияет на сет. Сеты обладают множеством методов для работы с уникальными элементами, например difference - возвращает элементы сета отсутствующие в другом сете, intersection - наоборот, возвращает элементы сета присутствующие в другом сете.

```
>>> s = set()
>>> print (s)
set()
>>> s = {"hi", "bye"}
>>> print (s)
{'hi', 'bye'}
>>> a = set()
>>> a.add(1)
>>> a.add(2)
>>> print (a)
{1, 2}
>>> a.add(2)
>>> print (a)
{1, 2}
>>> b = {2, 3}
>>> a.difference(b)
{1}
>>> b.difference(a)
{3}
>>> a.intersection(b)
{2}
```

Преобразование типов данных

Функция	Описание
int(x [,base])	Преобразовывает x в целое число. Например, int(12.4) -> 12
long(x [,base])	Преобразовывает x в long. Например, long(20) -> 20L
float(x)	Преобразовывает x в число с плавающей точкой. Например float(10) -> 10.0
complex(real [,imag])	Создает комплексное число. Например complex(20) -> (20+0j)
str(x)	Преобразовывает x в строку. Например str(10) -> '10'
tuple(s)	Преобразовывает s в кортеж. Например tuple("hello") -> ("h","e","l","l","o")
list(s)	Преобразовывает s в список. Например list("Python") -> ["P","y","t","h","o","n"]
dict(d)	Создает словарь из d. Например dict([(1,2), (3,4)]) -> { 1:2, 3:4 }

Функции в Python

Функция - это блок организованного, многократно используемого кода, который используется для выполнения конкретного задания. Функции обеспечивают лучшую модульность приложения и значительно повышают уровень повторного использования кода.

Создание функции

Существуют некоторые правила для создания **функций в Python**.

- Блок функции начинается с ключевого слова **def**, после которого следуют название функции и круглые скобки (()).
- Любые аргументы, которые принимает функция должны находиться внутри этих скобок.
- После скобок идет двоеточие (:) и с новой строки с отступом начинается тело функции.

```
1  def my_function(argument):
2      print argument
```

Вызов функции

После создания функции, ее можно исполнять вызывая из другой функции или напрямую из оболочки **Python**. Для вызова функции следует ввести ее имя и добавить скобки.

```
1 my_function("abracadabra")
```

Аргументы функции

Вызывая функцию, мы можем передавать ей следующие типы аргументов:

- *Обязательные аргументы (Required arguments)*
- *Аргументы-ключевые слова (Keyword argument)*
- *Аргументы по умолчанию (Default argument)*
- *Аргументы произвольной длины (Variable-length arguments)*

Обязательные аргументы

Если при создании функции мы указали количество передаваемых ей аргументов и их порядок, то и вызывать ее мы должны с тем же количеством аргументов, заданных в нужном порядке.

```
1  def bigger(a,b):
2      if a > b:
3          print a
4      else:
5         print b
6
7  # В описании функции указано, что она принимает 2 аргумента
8
9  # Корректное использование функции
10 bigger(5,6)
11
12 # Некорректное использование функции
13 bigger()
14 bigger(3)
15 bigger(12,7,3)
```

Аргументы – ключевые слова

Аргументы - ключевые слова используются при вызове функции. Благодаря ключевым аргументам, вы можете задавать произвольный (то есть не такой каким он описан при создании функции) порядок аргументов.

```
def person(name, age):
    print name, "is", age, "years old"

# Хотя в описании функции первым аргументом идет имя, мы можем вызвать функцию вот так

person(age=23, name="John")
```

Аргументы, заданные по умолчанию

Аргумент по умолчанию, это аргумент, значение для которого задано изначально, при создании функции.

```
1  def space(planet_name, center="Star"):
2      print planet_name, "is orbiting a", center
3
4  # Можно вызвать функцию space так:
5  space("Mars")
6  # В результате получим: Mars is orbiting a Star
7
8  # Можно вызвать функцию space иначе:
9  space("Mars", "Black Hole")
10 # В результате получим: Mars is orbiting a Black Hole
```

Аргументы произвольной длины

Иногда возникает ситуация, когда вы заранее не знаете, какое количество аргументов будет необходимо принять функции. В этом случае следует использовать аргументы произвольной длины. Они задаются произвольным именем переменной, перед которой ставится звездочка (*).

```
1  def unknown(*args):
2      for argument in args:
3          print argument
4
5  unknown("hello","world") # напечатает оба слова, каждое с новой строки
6  unknown(1,2,3,4,5) # напечатает все числа, каждое с новой строки
7  unknown() # ничего не выведет
```

Ключевое слово return

Выражение return прекращает выполнение функции и возвращает указанное после выражения значение. Выражение return без аргументов это то же самое, что и выражение **return None**. Соответственно, теперь становится возможным, например, присваивать результат выполнения функции какой либо переменной.

```
def bigger(a,b):
    if a > b:
        return a # Если a больше чем b, то возвращаем a и прекращаем выполнение функции
    return b # Незачем использовать else. Если мы дошли до этой строки, то b, точно не меньше чем a

# присваиваем результат функции bigger переменной num
num = bigger(23,42)
```

Область видимости

Некоторые переменные скрипта могут быть недоступны некоторым областям программы. Все зависит от того, где вы объявили эти переменные.

В **Python** две базовых области видимости переменных:

- *Глобальные переменные*
- *Локальные переменные*

Локальные переменные

Переменные объявленные внутри тела функции имеют локальную область видимости, те что объявлены вне какой-либо функции имеют глобальную область видимости.

Это означает, что доступ к локальным переменным имеют только те функции, в которых они были объявлены, в то время как доступ к глобальным переменным можно получить по всей программе в любой функции.

```
1   # глобальная переменная age
2   age = 44
3
4   def info():
5       print age # Печатаем глобальную переменную age
6
7   def local_info():
8       age = 22 # создаем локальную переменную age
9       print age
10
11  info() # напечатает 44
12  local_info() # напечатает 22
```

Глобальные переменные

Важно помнить, что для того чтобы получить доступ к глобальной переменной, достаточно лишь указать ее имя. Однако, если перед нами стоит задача *изменить* глобальную переменную внутри функции - необходимо использовать ключевое слово **global**.

```
1  # глобальная переменная age
2  age = 13
3
4  # функция изменяющая глобальную переменную
5  def get_older():
6      global age
7      age += 1
8
9  print age # напечатает 13
10 get_older() # увеличиваем age на 1
11 print age # напечатает 14
```

Рекурсия

Рекурсией в программировании называется ситуация, в которой функция вызывает саму себя. Классическим примером рекурсии может послужить функция вычисления факториала числа.

Напомним, что факториалом числа, например, 5 является произведение всех натуральных (целых) чисел от 1 до 5. То есть, 1 * 2 * 3 * 4 * 5

Рекурсивная функция вычисления факториала на языке Python будет выглядеть следующим образом:

```
1  def fact(num):
2      if num == 0:
3          return 1 # По договоренности факториал нуля равен единице
4      else:
5          return num * fact(num - 1) # возвращаем результат произведения num и результа
```

Массивы данных

Модуль array определяет массивы в python. Массивы очень похожи на списки, но с ограничением на тип данных и размер каждого элемента.

Класс **array.array**(TypeCode, [инициализатор]) - новый массив, элементы которого ограничены TypeCode, и инициализатор, который должен быть списком, объектом, который поддерживает интерфейс буфера, или итерируемый объект.

array.typecodes - строка, содержащая все возможные типы в массиве.

Массивы изменяемы. Массивы поддерживают все списковые методы (индексация, срезы, умножения, итерации), и другие методы.

Методы массивов в Python

array.typecode - TypeCode символ, использованный при создании массива.
array.itemsize - размер в байтах одного элемента в массиве.
array.append(x) - добавление элемента в конец массива.
array.buffer_info() - кортеж (ячейка памяти, длина). Полезно для низкоуровневых операций.
array.byteswap() - изменить порядок следования байтов в каждом элементе массива. Полезно при чтении данных из файла, написанного на машине с другим порядком байтов.
array.count(x) - возвращает количество вхождений x в массив.
array.extend(iter) - добавление элементов из объекта в массив.
array.frombytes(b) - делает массив array из массива байт. Количество байт должно быть кратно размеру одного элемента в массиве.

array.fromfile(F, N) - читает N элементов из файла и добавляет их в конец массива. Файл должен быть открыт на бинарное чтение. Если доступно меньше N элементов, генерируется исключение EOFError , но элементы, которые были доступны, добавляются в массив.
array.fromlist(список) - добавление элементов из списка.
array.index(x) - номер первого вхождения x в массив.
array.insert(n, x) - включить новый пункт со значением x в массиве перед номером n. Отрицательные значения рассматриваются относительно конца массива.
array.pop(i) - удаляет i-ый элемент из массива и возвращает его. По умолчанию удаляется последний элемент.
array.remove(x) - удалить первое вхождение x из массива.
array.reverse() - обратный порядок элементов в массиве.
array.tobytes() - преобразование к байтам.
array.tofile(f) - запись массива в открытый файл.
array.tolist() - преобразование массива в список.

План следующего занятия

- Двумерные массивы
- Вложенные списки и массивы
- Объектно-ориентированное и функциональное программирование на Python

Спасибо за внимание!