Лекция №1. Повторение. Введение в классы.

Материал лекции:

1. Строки
2. Списки
3. Кортежи
4. Словари
5. Функции
6. Классы

Строки

Строки – один из основных типов данных. С ними можно работать множеством различных способов, несмотря на простату этого типа данных.

Строка представляет собой простую последовательность символов. Любая последовательность символов, заключенная в кавычки, в Python считается строкой. При этом строки могут быть заключены как в одиночные, так и в двойные кавычки.

Пример:

```
print("При написании строки можно использовать внутренние кавычки и
апострофы:\n\tPython's strengths...")
```

Вывод:

>> При написании строки можно использовать внутренние кавычки и апострофы:
>> Python's strengths...

Изменение регистра символов в строках

```
first_name = "ada"
last_name = "lovELace  "
full_name = f"{first_name}{last_name.strip()}"
message = f"Hello, {full_name.title()}!"
print(message)
```

Метод title() представляет собой действие, которое Python выполнит с данными. Точка (.) после full_name в конструкции full_name.title() приказывает применить метод title() к переменной full_name. За именем метода всегда следует пара круглых скобок, потому что методам для выполнения их работы часто требуется дополнительная информация. Эта информация указывается в скобках. Методу title() дополнительная информация не нужна, поэтому в круглых скобках ничего нет.

Метод **title**() преобразует первый символ каждого слова в строке к верхнему регистру, тогда как все остальные символы выводятся в нижнем регистре.

В Python можно искать и удалять лишние пропуски у левого и правого края строки. Для этого нужно воспользоваться методом **rstrip**(), **lstrip**(), **strip**().

Списки

Список представляет собой набор элементов, следующих в определенной последовательности. В список можно поместить любую информацию, причем данные в списке не обязаны быть связаны друг с другом. Так как список содержит более одного элемента, рекомендуется присваивать спискам имена во множественном числе: **letters**, **digits**, **names** и т.д.

В языке Python список обозначается квадратными скобками ([]), а отдельные элементы списка отделяются запятыми.

Пример:

```
names = ["David", "rUso", "Monty"]
print(names)
names.append("Ben") #Добавить в конец строки элемент
names.insert(1, "Kay") #Добавить позицию элемент
del names[0] #Удалить элемент из позиции
print(names)
print(names[0]) #Обращение к нулевому индексу
for name in names:
    print(name.title())
popped_names = names.pop() #Удаляем элемент и записываем его в новую
переменную
print(popped_names)
```

Выводит на экран:

>> ['David', 'rUso', 'Monty']
>> ['Kay', 'rUso', 'Monty', 'Ben']
>> Kay
>> Kay
>> Ruso
>> Monty
>> Ben
>> Ben

Python считает, что первый элемент списка находится на позиции 0, а не в позиции 1. Второму элементу списка соответствует индекс 1.

- Метод **append**() присоединяет строку в конец списка, другие элементы списка остаются неизменными.
- Метод **insert**() добавляет новый элемент в произвольную позицию списка. Для этого нужно указать индекс и значение нового элемента.
- Команда **del** удаляет элемент из позиции и смещает список в ту или иную сторону.
- Метод **pop**() удаляет последний элемент списка, но позволяет запомнить удаленный элемент.
- Метод **remove**() удаляет элемент по значению.
- Метод **sort**() позволяет изменить, отсортировать и запомнить список в алфавитном порядке.
- Метод **sorted**() позволяет временно представить отсортированный список.
- Метод **len**() позволяет определить количество элементов в списке.

Кортежи

Списки хорошо подходят для хранения наборов элементов, которые могут изменяться на протяжении жизненного цикла программы. Например, возможность модификации списков жизненно необходима при работе со списками пользователей сайтов или списками персонажей игры. Однако в некоторых ситуациях требуется создать список элементов, который не изменяться. Кортежи (tuples) предоставляют именно такую возможность. В языке Python значения, которые не могут изменяться, называются неизменяемыми, а неизменяемый список называется кортежем.

Кортеж выглядит как список, не считая того, что вместо квадратных скобок используются круглые скобки. После определения кортежа вы можете обращаться к его отдельным элементам по индексам точно так же, как это делается при работе со списком.

Пример:

```
dimensions = (200, 50)
print(f'{dimensions} включает в себя: {dimensions[0]}, {dimensions[1]}')
```

Выводит на экран:

>> (200, 50) включает в себя: 200, 50

При попытке изменения кортежа программа выводит ошибку.

Пример:

```
…
dimensions[1] = 40
```

Вывод на экран:

```
Traceback (most recent call last):
  File "C:\*\Lection_1.py", line 3, in <module>
    dimensions[1] = 40
TypeError: 'tuple' object does not support item assignment
```

Словари

Словарь в языке Python представляет собой совокупность пар «ключ-значение». Каждый ключ связывается с некоторым значением, и программа может получить значение связанное с этим ключом. Значением может быть число, строка, список, другой словарь (или любой объект).

В Python словарь заключается в фигурные скобки {} в котором приводится последовательность пар «ключ-значение». Пара «ключ-значение» представляет данные, связанные друг с другом. Ключ отделяется от значения двоеточием (:), а отдельные пары запятыми. В словаре может хранить любое количество пары «ключ-значение».

Пример:

```
alien = {
        "color": "green",
        'points':5
}
new_points = alien['points'] #Извлекаем информацию о значение по ключу
print(f"You jast earned {new_points} points!")
alien["x_position"] = 0 #Добавляем пару «ключ-значение»
alien["y_position"] = 25 #Добавляем пару «ключ-значение»
print(alien)
```

Вывод на экран:

>> You jast earned 5 points!
>> {'color': 'green', 'points': 5, 'x_position': 0, 'y_position': 25}

Словарь относится к динамическим структурам данных: в словарь можно в любой момент добавлять новые пары «ключ-значение». Для этого указывается имя словаря, за которым в квадратных скобках следует новый ключ и через знак равно (=) новое значение.

В некоторых ситуациях удобнее создавать пустой словарь и добавлять в него новые элементы.

*Чтобы начать заполнение пустого словаря, определите словарь с пустой парой фигурных скобок, а затем добавляйте новые пары «ключ-значение».

Чтобы ***изменить значение*** в словаре, указывается имя словаря с ключом в квадратных скобках, а затем новое значение, которое должно быть связано с этим ключом.

Пример:

```
…
alien["color"] = "yellow"
print(alien)
```

Выводит на экран:

>> {'color': 'yellow', 'points': 5, 'x_position': 0, 'y_position': 25}

Для ***удаления*** пары «ключ-значение» используется команда del.

Пример:

```
…
del alien["points"]
print(alien)
```

Выводит на экран:

>> {'color': 'yellow', 'x_position': 0, 'y_position': 25}

Использование синтаксиса с ключом в квадратных скобках для получения интересующего вас значения из словаря имеет недостаток: если запрашиваемый ключ отсутствует в словаре, то выводится ошибка. В этом случае можно использовать метод **get**() для указания значения по умолчанию.

Пример:

```
…
print(alien.get("points", "No value"))
```

Выводит на экран:

>> No value

Словарь может содержать несколько пар «ключ-значение», поэтому в Python предусмотрены разные способы перебора словаря. Программа может перебрать все пары в словаре, только ключи или только значение.

Пример:

```
for key, value in alien.items():
    print(f"Key: {key}")
    print(f"Value: {value}")
print()
for name in alien.keys():
    print(name)
print()
for value in alien.values():
    print(value)
```

Выводит на экран:

>> Key: color
>> Value: yellow
>> Key: x_position
>> Value: 0
>> Key: y_position
>> Value: 25
>>
>> color
>> x_position
>> y_position
>>
>> yellow
>> 0
>> 25

- Метод **items**() возвращает список пар «ключ-значение».
- Метод **keys**() нужен для работы с ключами значений.
- Метод **values**() получает список значений без ключей.

Вместо того, чтобы помещать словарь в список, иногда удобнее помещать список в словарь.

Пример:

```
pizza = {
    "crast": "thick",
    "toppings": ["mushrooms", "extra cheese"]
}
for toppings in pizza["toppings"]:
    print(toppings)
```

Выводит на экран:

>> mushrooms
>> extra cheese

Словарь также можно вложить в другой словарь, но в таких случаях код усложняется. Например, если на сайте есть несколько пользователей с уникальными именами, вы можете использовать имена пользователей как ключи в словаре.

Пример:

```
users = {
    'one': {
        'first': 'Albert',
        'last': 'Einstein'
    },
    'two': {
        "first": "Marie",
        "last": "Curie"
    }
}
for username, user_info in users.items():
    print(username, user_info)
```

Выводит на экран:

>> one {'first': 'Albert', 'last': 'Einstein'}
>> two {'first': 'Marie', 'last': 'Curie'}

Функция

Функции – именованные блоки кода, предназначены для решения одной конкретной задачи. Чтобы выполнить задачу, определенную в виде функции, вы вызываете функцию, отвечающую за задачу. Если задача должна многократно выполняться в программе, вызывается функция, предназначенная для этого значения, а этот вызов прикажет Python выполнить код, содержащийся внутри функции.

Запись:

```
def имя_функции(параметр_1, параметр_2, параметр_3, параметр_4):
    тело_функции
```

Пустая функция:

```
def empty_def():
    pass
empty_def()
```

Существует два основных вида аргументов, которые можно передавать в функцию.

- *Именованный аргумент* – пара «имя – значение».
- *Позиционный аргумент* – указывается только значение

Пример:

```
def great_user(name = "None_name"): #Указали значение и необязательный
аргумент для функции
    print(f'Hello, {name}')
users = {'One': 'Albert', 'Two': 'Marie'}
for name in users.values():
    great_user(name) #Вызываем функцию great_user с указанием значения name
great_user('Frank') #Вызов функции с указанием позиционного аргумента
great_user() #Используется необязательный аргумент функции
```

Выводит на экран:

>> Hello, Albert
>> Hello, Marie
>> Hello, Frank
>> Hello, None_name

Классы

Объектно-ориентированное программирование (в дальнейшем используется аббревиатура - ООП) по праву считается одной из самых эффективных методологий создания программных продуктов. В ООП создаются классы, описывающие реально существующие предметы и ситуации, а затем создаются объекты на основе этих описаний. При написании класса определяется общее поведение для целой категории объектов.

Когда вы создаете конкретные объекты на базе этих классов, каждый объект автоматически наделяется общим поведением. После этого вы можете наделить каждый объект уникальными особенностями на свой выбор.

Создание объекта на основе класса называется *создание экземпляра*, таким образом, вы работаете с экземплярами класса.

Классы позволяются моделировать практически что угодно. Создадим простой класс Dog. В этот класс будут включены следующие значения: кличка, возраст, некоторые команды. После того, как класс будет написан, мы используем его для создания экземпляров, каждый из которых представляет одну конкретную собаку.

Пример:

```
class Dog():
    def __init__(self, name, age): #Метод
        self.name_gl = name
        self.age_gl = age
    def command_sit(self):
        name = self.name_gl
        print(f'{name} - sit!')
    def command_roll_over(self):
        print(f"{self.name_gl} rolled over!")

my_dog = Dog("Charly", 6) #Экземпляр класса
print(f'My dog is - {my_dog.name_gl}') #Обращение к атрибуту
my_dog.command_sit()
my_dog.command_roll_over()
```

Выводит на экран:

>> My dog is - Charly
>> Charly - sit!
>> Charly rolled over!

По общепринятым соглашениям имена, начинающиеся с символа верхнего регистра, в Python обозначаются классами. Круглые скобки в определении класса пуста, потому что класс создается с нуля.

Функция, являющаяся частью класса, называется методом.

Метод __init__ - специальный метод, который автоматически выполняется при создании нового экземпляра на базе класса Dog. Имя метода начинается и заканчивается двумя символами подчеркивания, эта схема предотвращает конфликты имен стандартных методов Python и методов ваших классов.

!!! Два символа подчеркивания должны стоять на каждой стороне __init__(). Если вы поставите только один символ подчеркивания с каждой стороны, то метод не будет вызываться автоматически при использовании класса, что может привести к появлению ошибок.

Метод __init__() определяется с тремя параметрами: self, name, age. Параметр self обязателен в определении метода, он должен предшествовать всем остальным параметрам и должен быть включен в определение, потому что при будущем вызове метода __init__() (для создания экземпляра Dog) Python автоматически передает аргумент self. При каждом вызове метода, связанного с классом, автоматически передается self – ссылка на экземпляр, она предоставляет конкретному экземпляру доступ к атрибутам и методам класса.

Когда вы создаете экземпляр Dog, Python вызывает метод __init__() из класса Dog. Мы передаем Dog() кличку и возраст в аргументах, значение self передается автоматически, так, что его передавать не нужно. Каждый раз, когда вы захотите создать экземпляр на основе класса Dog, необходимо предоставить значения только двух последних аргументов name и age.

Каждая из двух переменных:

```
self.name_gl = name
self.age_gl = age
```

Снабжена префиксом self. Любая переменная с префиксом self доступна для каждого метода в классе, и вы также можете обращаться к этим переменным в каждом экземпляре, созданном на основе класса. Конструкция: self.name_gl = name берет значение, хранящееся в параметре name, и сохраняет его в переменной name_gl, которая затем связывается с создаваемым экземпляром.

Переменные, к которым вы обращаетесь через экземпляры, также называются атрибутами.

В классе Dog также определяются два метода command_sit и command_roll_over. Так как этим методам не нужна дополнительная информация, они определяются с единственным параметром self. Экземпляры могут вызывать этот метод:

```
my_dog = Dog("Charly", 6)
print(f'My dog is - {my_dog.name_gl}')
my_dog.command_sit()
my_dog.command_roll_over()
```

Для обращения к атрибутам экземпляра используется точечная запись. В строке my_dog.name_gl мы обращаемся к значению атрибута name_gl экземпляра my_dog.

Чтобы вызвать метод, укажите экземпляр(my_dog) и вызываемый метод, разделив их точкой. В ходе обработки my_dog.command_sit() Python ищет метод command_sit() в классе Dog() и выполняет его код.

На основе класса можно создать столько экземпляров, сколько вам потребуется.

Пример:

```
your_dog = Dog("Jesy", 9)
print(f"Your dog - {your_dog.name_gl}")
```

Выведет на экран:

>> Your dog – Jesy

Классы могут использоваться для моделирования многих реальных ситуаций. После того, как класс будет написан, разработчик проводит большую часть времени за работой с экземплярами, созданными на основе этого класса. Одной из первых задач станет изменение атрибутов связанных с конкретным экземпляром. Атрибуты экземпляра можно изменять напрямую или же написать методы, изменяющие атрибуты по особым правилам.

Почта: **suhanova@mer.ci.nsu.ru**

Ссылка на тест: https://forms.gle/q5GEdax4aaMgfY539