Урок 3

Стандартная библиотека Python

Базовые функции. Работа с списками. Кортежи, словари, встроенные функции. Стандартные модули Python.

Базовые операции со строками

- ***Какие базовые операции сложения строк вы знаете?***

```
def str_handling():
    # 1. Сложение строк:
    print('Hello' + ' ' + 'world')

    # 2. Если строки идут друг за другом, + можно опустить
    #(конкатенация строк произойдет автоматически):
    print('Hello'  ' '  'world')

    # 3. Строки можно повторять операцией *:
    print('Hey! ' * 3)

    # 4. Получение символа строки по индексу(каждая строка
    # это неизменяемый массив char):
    # Все элементы строки нумеруются порядковыми индексами
    #(первый индекс НОЛЬ):
    string = 'произвольная строка'
    print('string = ', string)
    print('string[1] --> ', string[1])

    # 5. Срезы
    # Подстроку можно получить при помощи срезов:
    print('string[6:11] -->', string[6:11])

    # Значения по умолчанию: опущенный первый индекс заменяется нулем,
    # опущенный второй индекс подменяется размером срезаемой строки.
    print('string[6:] -->', string[6:])
    print('string[:11] -->', string[:11])

    # Чересчур большой индекс заменяется на размер строки:
    print(string[6:100])

    # Верхняя граница меньше нижней возвращает пустую строку:
    print('string[50:] -->', string[50:])
    print('string[6:1] --> ', string[6:1])

    # Индексы могут быть отрицательными числами,
    # обозначая при этом отсчет справа налево:
    print('string[-1] -->', string[-1])     # Последний символ
    print('string[-2] -->', string[-2])     # Предпоследний символ
    print('string[-2:] -->', string[-2:])   # Последние два символа
    print('string[:-2] -->', string[:-2])# Все, кроме последних двух символов

    # Хороший способ понять, как работают срезы -
    # думать о них, как об указателях на места между символами:
    #  +---+---+---+---+---+
    #  | L | o | r | e | m |
    #  +---+---+---+---+---+
```

```
    #  0   1   2   3   4   5
    # -5  -4  -3  -2  -1

    # 6. Срезы с шагом
    # Получаем каждый второй символ для указанного среза
    print('string[:12:2] -->', string[:12:2])
    # Переворачиваем строку задом наперед
    print('string[::-1] -->', string[::-1])

    # 7. Длина строки:
    print(len(string))

str_handling()
```

Методы строк

- ***Какие методы для строк вы знаете?***

С помощью методов мы можем выполнить действия с объектом. Метод вызывается через точку после объекта.

При вызове методов необходимо помнить, что строки в Python относятся к категории неизменяемых последовательностей, то есть все функции и методы могут лишь создавать новую строку.

```
def str_methods():
    print("Устанавливаем первую букву слова заглавной: " +
                                                    'программа'.title())

    print("Устанавливаем первую букву слова заглавной: " +
                                                    'программа'.upper())

    print("Номер первого вхождения подстроки 'ра' в строку 'разработка' - " +
                                                str('разработка'.find('ра')))

    print("Номер последнего вхождения подстроки 'ра' в строку 'разработка' - "
                                            + str('разработка'.rfind('ра')))

    print("Номер вхождения подстроки 'ра' в строку 'разработка' с указанием
    индекса начала поиска - "
                                            + str('разработка'.find('ра', 2)))

str_methods()
```

С помощью метода **.title()** первая буква в строке становится заглавной. Посредством метода **.upper()** все символы в строке переходят в верхний регистр. Существуют методы, которые принимают дополнительные аргументы — например, **find()**, отвечающий за поиск определенной подстроки в строке. Он возвращает номер последнего вхождения или -1. Вторым аргументом принимает индекс начала поиска (по умолчанию поиск производится с начала строки).

Подробнее о методах здесь.

Внимание! Методы можно применять и к переменной, и к значению. В любом случае метод применяется к самой строке.

Форматирование строк

- ***Как можно форматировать строки в Python?***

Довольно часто возникают ситуации, когда нужно сделать строку, подставив в нее данные, полученные при выполнении программы (пользовательский ввод, данные из файлов и подобное).

Делать это, как в примере ниже, не рекомендуется, так как ухудшается читаемость, а запись сложно редактировать.

```
def bad_way():
    name = 'Иван'
    surname = 'Иванов'
    print('Welcome, ' + surname + ' ' + name + ', to our conference')

bad_way()
```

Подстановку данных можно выполнить с помощью форматирования строк с помощью оператора %:

```
def good_way(name, surname):
    # Старый способ форматирования
    print('Welcome, %s %s, to our conference' % (name, surname))

good_way('Иван', 'Иванов')
```

Или с помощью метода **format**:

```
def best_way(name, surname):
    # Более новый и  гибкий метод
    print('Welcome, {} {}, to our conference'.format(name, surname))
    print('Welcome, {1} {0}, to our conference'.format(name, surname))

best_way('Иван', 'Иванов')
```

Метод **.format()** — наиболее гибкий и имеет много возможностей для форматирования. Фигурными скобками { } указываем места в строке-шаблоне, куда будет выполняться подстановка данных. Цифрами {0} {1} можем изменить порядок подставляемых данных.

Примеры:

Именованные параметры

```
def named_parameters():
    text = "Hello, {first_name}.".format(first_name="Tom")
    print(text)    # Hello, Tom.

    info = "Name: {name} Age: {age}".format(name="Bob", age=23)
    print(info)    # Name: Bob  Age: 23

named_parameters()
```

Параметры по позиции

```
def position_parameters():
    info = "Name: {0}\t Age: {1}".format("Bob", 23)
    print(info)    # Name: Bob  Age: 23

position_parameters()
```

Подстановки

- **s**: для вставки строк;
- **d**: для вставки целых чисел;
- **f**: для вставки дробных чисел. Для этого типа также можно определить через точку количество знаков в дробной части;
- **%**: умножает значение на 100 и добавляет знак процента;
- **e**: выводит число в экспоненциальной записи.

```
def substitution():
    number = 23.8589578
    print("{:.2f}".format(number))    # 23.86
    print("{:.3f}".format(number))    # 23.859
    print("{:.4f}".format(number))    # 23.8590

substitution()
```

Познакомиться со всеми инструментами форматирования можно здесь.

Списки

- ***Что такое списки в Python?***

Списки в Python — это неограниченные по размеру коллекции произвольных объектов. Списки являются изменяемыми объектами, к ним можно применить операции как с помощью индексов, так и методов работы со списками.

Так как списки являются последовательностью, они поддерживают все операции, которые можно применить к строкам. Отличие лишь в том, что результатом этих операций будут списки, а не строки.

Если строки создаются с помощью литералов кавычек “” / ‘’, то списки создаются литералами квадратных скобок [ ].

Пример операций со списками:

```
def list_handling():

    # Список - изменяемая последовательность, элементы которой - любые типы
данных
    empty_list = []        # пустой список
    my_list = [1, 3, 5, 3.45, 'ddd', 's', 333]
    print('my_list = ', my_list)

    # Т.к. список - это последовательность, к нему применимы те же операции
    #, как к строке.
    # Получение элемента по индексу
    print(my_list[0])      # получим первый элемент списка
    print(my_list[-1])     # последний элемент списка

    # Срезы
    print(my_list[0:-3])  # 3 последних элемента списка

    # Конкатенация
    print(my_list[0:3] + [7, 8, 9])  # получим новый список из 6 элементов

    # Мультипликация
    print([3, '4'] * 3)    # размножим список

    # В отличие от строк, элементы списка можно изменять:
    my_list[2] = 'New'
    print('my_list after change =', my_list)

    # А также заменять часть элементов с помощью срезов. Заменим первые 3:
    my_list[0:3] = [2, 4, 6]
    print(my_list)

    # Удалим последние 2
    my_list[-2:] = []
    print(my_list)

    # Вставим несколько элементов внутрь
    my_list[3:3] = ['this', 'is', 'some', 'elements']
    print(my_list)

    # Вставим элемент в начало списка
    my_list[:0] = ['first']
    print(my_list)
```

```
    # Как и для строк, встроенная функция len() вернет длину списка:
    print(len(my_list))

    # Добавить что-то в конец списка можно так:
    my_list[len(my_list):] = [100]
    print(my_list)

    # Но чаще используется более простая конструкция
    # (простое лучше сложного, правда?):
    my_list.append(200)
    print(my_list)

    # Можно создавать списки, содержащие другие списки:
    b = [1, 2, 3, [11, 22, 33], 5, 6]
    print('b = ', b)
    print('b[3][2] =', b[3][2])

    # Оператор вхождения in
    print('2 in b -->', 2 in b)
    print("'2' in b -->", '2' in b)

list_handling()
```

Методы списков

- ***Какие методы у списков вы знаете?***

Список, по сути, — это изменяемый объект, который можно модифицировать многими методами.

```
def list_methods():
    lst = [1, -2]
    # добавит элемент в конец списка
    lst.append(4)
    print(lst)
    # удалит последний элемент списка и вернет его
    lst.pop()
    print(lst)
    # удалит элемент списка с индексом 1
    lst.pop(1)
    print(lst)

list_methods()
```

Кортежи

- ***Что такое кортежи в Python?***

Кортеж — неизменяемый список. ***Зачем нужны кортежи***, если есть списки?

1. «Защита от дурака». Кортеж защищен от изменений — как намеренных (что плохо), так и случайных (что хорошо).
2. Меньший размер — по сравнению со списками при одинаковом количестве элементов.

Создать кортеж можно с помощью круглых скобок:

```
t = ()
```

Чтобы создать кортеж, состоящий из одного элемента, нужно после него поставить запятую:

```
def tuple_gener():
    # так не годится, получим int
    t = (2)
    print(t)
    # а так получим именно кортеж
    t = (2, )
    print(t)
    # так тоже получим кортеж
    t = 2,
    print(t)

tuple_gener()
```

Таким образом, кортеж создается не самим наличием круглых скобок, а с помощью запятой.

К кортежам можно применять те же операции и методы, что и к спискам, за исключением тех, что меняют сам кортеж.

Словари

- ***Что такое словарь в Python?***

Словарь в языке Python — это отображения.

Отображения — это коллекции объектов, но доступ к ним осуществляется не по определенным смещениям от начала коллекции (индексам), а по ключам.

Словари — единственный тип отображения в наборе базовых объектов Python. Они относятся к классу изменяемых объектов: могут изменяться непосредственно, увеличиваться и уменьшаться в размерах, как списки.

Программный код определения словаря заключается в фигурные скобки и состоит из последовательности пар «ключ: значение».

Словари удобно использовать, когда возникает необходимость связать значения с ключами — например, чтобы описать свойства чего-либо:

```
fruit = {"name": "Carrot", "color": "orange", "quantity": 12}
```

Мы можем обращаться к элементам этого словаря по ключам и изменять значения, связанные с ключами. Для доступа к элементам словаря используется тот же синтаксис, что и для обращения к

элементам последовательностей — только в квадратных скобках указывается не смещение относительно начала последовательности, а ключ:

```
>>> fruit["name"]
: Carrot
```

При обращении к элементу словаря с несуществующим ключом возникнет ошибка:

```
>>> fruit["from"]
: ... KeyError: 'from'
```

Добавление значения в словарь происходит присваиванием значения несуществующему ключу:

```
>>> new_dict = {} # создаем пустой словарь
>>> new_dict["new"] = "value"
>>> print(new_dict)
: {"new": "Value"}
```

Присваивание нового значения по существующему ключу ***заменяет значение на новое***.

Методы словарей

- ***Какие методы есть у словарей?***

```
def dict_handling():
    # пример словаря
    fruit = {"name": "Carrot", "color": "orange", "quantity": 12}
    # цикл по словарю
    for key, value in fruit.items():
        print(key, value)
    # для перебора ключей
    for key in fruit.keys():
        print(key)
    # для перебора значений, соответствующих ключам, можно не указывать keys
    for value in fruit.values():
        print(value)
    # удаляет элемент с и возвращает его значение
    print(fruit.pop('name'))
    print(fruit)
    # удаляет первый элемент и возвращает пару (ключ, значение)
    print(fruit.popitem())

dict_handling()
```

Метод **.items()** — возвращает пары (ключ, значение).

Метод **.keys()** — возвращает список ключей.

Встроенные функции

- ***Какие встроенные функции вы знаете?***

Мы уже пользовались некоторыми встроенными функциями Python, которые решают наиболее часто возникающие задачи:

- **print()**;
- **input()**;
- **len()**;
- функциями преобразования типов **int()**, **float()**, **bool()** и т.д.

Кратко рассмотрим еще группу наиболее используемых функций:

- **range([start=0], stop, [step=1])** — арифметическая прогрессия от **start** до **stop** с шагом **step**;
- **abs(x)** — возвращает абсолютную величину (модуль числа);
- **max(iter, [args ...] * [, key])** — максимальный элемент последовательности;
- **min(iter, [args ...] * [, key])** — минимальный элемент последовательности;
- **round(X [, N])** — округление до N знаков после запятой;
- **sum(iter, start=0)** — сумма членов последовательности;
- **type(object)** — возвращает тип объекта;
- **enumerate(string)** — возвращает пары (элемент, его индекс).

Если принцип функции не очевиден из описания — поэкспериментируйте в оболочке интерпретатора или посмотрите примеры в проекте.

Обратите внимание! Функции **max()**, **min()**, **sum()**, **len()** принимают в качестве первого аргумента объект-итератор (любую последовательность).

Рассмотрим еще несколько встроенных функций.

zip()

Принцип работы этой функции проще показать на примерах.

```
def zip_example_1():
    a = [1,2]
    b = [3,4]
    # выводим объект - архив
    print(zip(a,b))
    # выводим объект в список
    print(list(zip(a,b)))

zip_example_1()
```

Выведет: **[(1, 3), (2, 4)]**.

```
def zip_example_2():
    a = [1,2,4]
    b = [3,4]
    c = [5,6,0]
    # выводим объект - архив
    print(zip(a,b,c))
    # выводим объект в список
    print(list(zip(a,b,c)))

zip_example_2()
```

Выведет:**[(1, 3, 5), (2, 4, 6)]**.

Берет по минимальному количеству элементов, остальные будут отброшены.

map()

Позволяет применить функцию к каждому элементу последовательности, а результаты функции возвращает в виде итератора.

Например, нам нужно возвести каждый элемент последовательности в квадрат:

```
print(list(map(lambda x: x*x, [2, 5, 12, -2])))
```

map(func_link, <итератор>) --> итератор, каждым элементом которого является применение функции **funk_link** к элементам исходного итератора.

Результат оборачиваем в **list()**, чтобы увидеть полный результат.

filter()

```
# filter(filter_func, <итератор>) --> итератор с отфильтровыванием элементов
функцией filter_func
print(list(filter(lambda x: x > 5, [2, 10, -10, 8, 2, 0, 14])))
# Отбрасываем все элементы длиной НОЛЬ
print(list(filter(len, ['', 'not null', 'bla', '', '10'])))
```

filter() отбрасывает те элементы, для которых функция возвращает **False**.

Работа с файлами

- ***Как работать с файлами?***

Прежде чем работать с файлом, его надо открыть. С этим справится встроенная функция **open()**.

```
import os
# не самый хороший способ задания пути:
path = 'files/text.txt'
# хороший кроссплатформенный метод указания пути:
path = os.path.join('files', 'text.txt')
f = open(path, 'r', encoding='UTF-8')
# Считываем всю информацию из файла в виде списка строк
print(f.readlines())
f.close()
```

encoding='UTF-8' — указываем кодировку файла. Без этой строки под Windows много проблем.

‘r’ — режим «на чтение».

Режимы работы с файлом:

Режим	**Обозначение**
‘r’	открытие на чтение (является значением по умолчанию)
‘w’	открытие на запись: содержимое файла удаляется; если файла не существует, создается новый
‘x’	открытие на запись, если файла не существует — иначе исключение
‘a’	открытие на дозапись, информация добавляется в конец файла
‘b’	открытие в двоичном режиме
‘+’	открытие на чтение и запись

Режимы могут быть объединены — к примеру, '**rb**' — чтение в двоичном режиме. По умолчанию режим равен '**rt**'.

При открытии файла **open()** возвращает итератор. Используя цикл **for in**, мы можем читать информацию построчно.

```
path = os.path.join('files', 'text.txt')
f = open(path, 'r', encoding='UTF-8')
wanted_symbol = "+"
for line in f:                  # считываем файл построчно
    if wanted_symbol in line:   # пока не найдем нужную информацию
        print(line)
        break                   # когда нашли, заканчиваем чтение файла
```

Рекомендуется работать с файлами, используя менеджер контекста **with**.

```
# Наиболее правильный способ работы с файлами
# По окончании инструкции with, файл гарантировано будет закрыт, даже если
произойдет ошибка
with open(path, 'r', encoding='UTF-8') as f:
    print(f.readlines())
```

Для записи информации в файл используется метод **.write()**.

```
with open('newfile.txt', 'w', encoding='utf-8') as g:
    d = int(input('Введите число: '))
    g.write(str(d))
```

Передача аргументов по ссылке/значению

- ***Как передать аргумент по ссылке/значению?***

Рассмотрим такие примеры:

```
def args_send():
    n1 = 2
    n2 = n1
    n2 = 4
    print("n1 = ", n1, "n2 = ", n2)
    sp1 = [1, 2, 3]
    sp2 = sp1
    sp2.append(4)
    print("sp1 = ", sp1, "sp2 = ", sp2)

args_send()
```

Результат:

```
n1 = 2 n2 = 4
sp1 = [1, 2, 3, 4] sp2 = [1, 2, 3, 4]
```

Почему список **sp1** тоже изменился, хотя мы изменяли только **sp2**?

Переменная в Python — это всего лишь указатель на объект в памяти. Если несколько переменных указывают на один и тот же объект, то, изменив его по одной из ссылок, мы меняем его и для всех остальных.

Это особенно важно понимать при передаче изменяемых объектов в функцию и при изменении объекта в цикле **for in** (который итерирует данный объект).

```
def modify(lst):
    lst.append("new")
    return lst

my_list = [1, 2, 3]
mod_list = modify(my_list)
# Функция вернула измененный список
print('mod_list = ', mod_list)
# Но исходный список тоже изменился, подобное неявное поведение нежелательно для функций
print('my_list = ', my_list)
```

Результат:

```
mod_list =  [1, 2, 3, 'new']
my_list =  [1, 2, 3, 'new']
```

Как защитить исходный список от модификации? В функцию изменения исходного списка в качестве аргумента следует передать ссылку на список следующим образом: **my_list[:]**. Пример:

```
def modify(lst):
    lst.append("new")
    return lst

my_list = [1, 2, 3]
mod_list = modify(my_list[:])
# Функция вернула измененный список
print('mod_list = ', mod_list)
# Исходный список теперь не изменился
print('my_list = ', my_list)
```

Результат:

```
mod_list =  [1, 2, 3, 'new']
my_list =  [1, 2, 3]
```

Рассмотрим изменение списка в процессе перебора его элементов циклом **for..in**.

```
my_list = [1, -2, -4, 0, 5, -2]
# Удаляем все отрицательные элементы
for el in my_list:
    if el < 0:
        my_list.remove(el)
# Вероятно, это не тот результат, который вы ожидали
print("1)my_list после удаления -->", my_list)
# 1) my_list after remove --> [1, -4, 0, 5]
my_list = [1, -2, -4, 0, 5, -2]
# Итерируем по копии, а удаляем из оригинала
for el in my_list[:]:
    if el < 0:
        my_list.remove(el)
# Как хорошо
print("2)my_list after remove -->", my_list)
# 2)my_list after remove --> [1, 0, 5]
```

Получается, что при переборе элементов списка те, что не удовлетворяют условию, удаляются не в полном объеме. Это происходит потому, что при удалении элемента на его место становится следующий. Но так как мы переходим к следующему элементу, не проверяем тот, что оказался на месте удаленного.

Подход с использованием копии списка позволяет решить эту проблему. Проверку условия мы осуществляем на копии списка и ничего в нем не удаляем, то есть элемент для удаления определяем в копии списка. А непосредственно удаление выполняем уже в исходном списке.

Генераторы списков и словарей

- ***Что такое генератор словарей и списков?***

Генераторы — яркий пример «синтаксического сахара» в Python. Все, что можно сделать с помощью генераторов, можно выполнить и без них. Это всего лишь синтаксическая конструкция, позволяющая записать частые операции красиво и кратко. Но все же надо учесть, что генераторы обрабатывают результаты быстрее, чем аналогичные им конструкции, реализованные через циклы **for in**.

```
import random
# Заполняем список произвольными целыми числами
lst = []
for _ in range(10):
    lst.append(random.randint(-10, 10))

print('lst = ', lst)
# То же самое, но с помощью генератора списка
# Компактнее код и выполняется быстрее
lst_g = [random.randint(-10, 10) for _ in range(10)]
print('lst_g = ', lst_g)
```

По сути, генератор — это свернутый цикл **for in**.

Обратили внимание на странное имя переменной **_** (нижнее подчеркивание)? Если переменная должна присутствовать в соответствии с синтаксисом, но ее значение нигде не используется, принято такую переменную называть **_** (нижнее подчеркивание) — это общепринятое соглашение.

```
# Отбрасываем все отрицательные элементы списка
only_positive = [el for el in lst_g if el >= 0]
print('only_positive = ', only_positive)
```

lst_g — список, заполненный в предыдущем примере.

Элемент добавляется в список, если выражение после **if** — **True**.

Аналогично можно создавать словари с помощью генераторов.

```
# Создаем словарь с помощью генератора словаря
keys = "abcdefg"
values = range(10)
dict_g = {key: value for key, value in zip(keys, values)}

# О функции zip сказано выше
print('dict_g = ', dict_g)

# Более простой пример создания словаря генератором
dict2_g = {el: el+4 for el in [1, 4, 6, 8]}
print('dict2_g =', dict2_g)
```

Получим результаты:

```
dict_g =  {'a': 0, 'd': 3, 'g': 6, 'f': 5, 'e': 4, 'c': 2, 'b': 1}
dict2_g = {8: 12, 1: 5, 4: 8, 6: 10}
set_g = {0, 1, 'g', 'b', 'a', -10}
```

Генераторы — очень удобные и мощные инструменты. Важно понимать, что генератор — это выражение, которое в качестве результата возвращает определенную последовательность. Следовательно, можно использовать генераторы внутри других выражений. Но не злоупотребляйте, иначе получите код, сложный для чтения и редактирования.

Регулярные выражения

- ***Что такое регулярные выражения? Как использовать регулярные выражения?***

Регулярные выражения по сути являются последовательностью символов, используемых для поиска и замены в строке или файле. Они встречаются практически во всех языках программирования.

Регулярные выражения могут использовать два типа символов:

- **специальные символы**: например, ***** означает «любой символ»;
- **литералы**: например, **a**, **b**, **1**, **2** и т. д.

Чаще всего регулярные выражения применяются для поиска подстроки в строке, разбиения строки, замены части строки.

В Python для работы с регулярными выражениями есть модуль **re**. Для использования его нужно импортировать.

re.match(pattern, string)

С помощью этой функции можно проверить, соответствует ли начало строки "**string**" регулярному выражению "**pattern**".

К примеру:

```
string = 'This is a simple test message for test'
string2 = 'test'
pattern1 = 'test$'
pattern2 = '^test'
pattern3 = '^test$'
print(re.search(pattern1, string) is None)    # Строка заканчивается на 'test'
print(re.match(pattern2, string) is None)     # Строка не начинается на 'test'
print(re.match(pattern3, string) is None)     # Строка не является строкой 'test'
print(re.match(pattern3, string2) is None)  # Строка является строкой 'test'
```

Получим:

```
False
True
True
False
```

re.search() — работает аналогично **re.match**, но полностью проверяет всю строку на совпадение.

re.findall() — с помощью этой функции можно найти все вхождения подстрок, соответствующих регулярному выражению.

К примеру:

```
# Сколько раз слово присутствует в строке
string = 'This is a simple test message for test'
found = re.findall(r'test', string)
print(found)
```

В результате:

```
['test', 'test']
```

Еще есть методы:

- **re.split()** — разделяет строку по заданному шаблону;
- **re.sub()** — ищет шаблон в строке и заменяет его на указанную подстроку;

- **re.compile()** — собирает регулярное выражение в отдельный объект для поиска. Это позволяет использовать одно и то же выражение многократно.

Если у нас нет определенного шаблона для подстроки, можно написать свое выражение с использованием специальных символов.

Метасимвол	Описание
.	Один любой символ, кроме новой строки \n
?	0 или 1 вхождение шаблона слева
+	1 и более вхождений шаблона слева
*	0 и более вхождений шаблона слева
\w	Любая цифра или буква (\W — все, кроме буквы или цифры)
\d	Любая цифра [0-9] (\D — все, кроме цифры)
\s	Любой пробельный символ (\S — любой не пробельный символ)
\b	Граница слова
[..]	Один из символов в скобках ([^..] — любой символ, кроме тех, что в скобках)
\	Экранирование специальных символов (\. означает точку или \+ — знак «плюс»)
^ и $	Начало и конец строки соответственно
{n,m}	От n до m вхождений ({,m} — от 0 до m)
a | b	Соответствует a или b
()	Группирует выражение и возвращает найденный текст
\t, \n, \r	Символ табуляции, новой строки и возврата каретки соответственно

Пример:

```
# Найти все цифры в тексте
pattern = '[0-9]+k'
string = 'If 300 spartans were so brave, so 500 spartans' \
    ' could destroy more than 10k warriors of Darius, but 15k and even 20k'
print(re.findall(pattern, string))

# Найти все диапазоны
pattern2 = '[0-9]+ *- *[0-9]+'
string2 = 'The temperature can be in range 10- 15C next week ' \
          'though it was lesser last week(4 - 9C). It was even ' \
          '-5 some time ago'
print(re.findall(pattern2, string2))
```

В результате:

```
['10k', '15k', '20k']
['10- 15', '4 - 9']
```

Обработка исключений

- ***Как можно обработать исключения/ошибки?***

Все ошибки в Python являются типом данных. Исключения сообщают программисту об ошибках, возникших в результате выполнения программы.

Один из самых простых примеров — деление на ноль:

```
>>> 100 / 0
```

Для обработки исключений используется конструкция **try — except**.

```
# n = 10
n = 'Hello'
try:
    n = int(n)
    print('n успешно преобразована к типу Int')
except ValueError:  # Тип перехватываемого исключения
    print('значение n невозможно преобразовать к типу int')
```

После **try** мы выполняем инструкции, которые могут привести к ошибкам, а в **except** нужным образом их обрабатываем. Если исключений в результате работы программы не возникло, то все инструкции, написанные в блоке **except**, не будут выполнены.

Если у **except** нет аргументов, то эта инструкция перехватывает вообще все (прерывание с клавиатуры, системный выход и т.д.). Поэтому без аргументов эта функция почти не используется, а применяется **except Exception**.

Есть еще две инструкции, которые относятся к обработке ошибок — это **finally** и **else**. **Finally** выполнит блок с инструкциями в любом случае, независимо от того, возникло ли исключение (применяется, когда нужно непременно что-то сделать — к примеру, закрыть файл). Код, идущий после **else**, будет выполнен, если исключение не сгенерировано.

```
f = open('1.txt')
ints = []
try:
   for line in f:
      ints.append(int(line))
except ValueError:
      print('Это не число. Выходим.')
except Exception:
      print('Это что еще такое?')
else:
       print('Все хорошо.')
finally:
      f.close()
      print('Я закрыл файл.')
# Именно в таком порядке: try, группа except, затем else, и только потом finally.
```

Модуль OS

- ***Какие функции из модуля OS вы знаете?***

Модуль **OS** предоставляет множество функций для работы с операционной системой, причем их поведение, как правило, не зависит от ОС, поэтому программы остаются переносимыми.

С помощью функций модуля **OS** можно узнать тип операционной системы:

```
import os
print('os.name = ', os.name)
```

Можно получить полный путь к запущенному файлу:

```
print('os.getcwd() = ', os.getcwd())
```

Создать новую директорию:

```
dir_path = os.path.join(os.getcwd(), 'NewDir')
try:
    os.mkdir(dir_path)
except FileExistsError:
    print('Такая директория уже существует')
```

Еще немного информации о функциях модуля **OS**:

- **os.chdir(path)** — смена текущей директории;
- **os.chmod(path, mode, *, dir_fd=None, follow_symlinks=True)** — смена прав доступа к объекту (**mode** — восьмеричное число);
- **os.chown(path, uid, gid, *, dir_fd=None, follow_symlinks=True)** — меняет id владельца и группы (Unix);

- **os.getcwd()** — текущая рабочая директория;
- **os.mkdir(path, mode=0o777, *, dir_fd=None)** — создает директорию. **OSError**, если директория существует;
- **os.remove(path, *, dir_fd=None)** — удаляет путь к файлу;
- **os.rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None)** —переименовывает файл или директорию из **src** в **dst**;
- **os.renames(old, new)** — переименовывает **old** в **new**, создавая промежуточные директории;
- **os.replace(src, dst, *, src_dir_fd=None, dst_dir_fd=None)** —переименовывает из **src** в **dst** с принудительной заменой;
- **os.rmdir(path, *, dir_fd=None)** — удаляет пустую директорию;
- **os.symlink(source, link_name, target_is_directory=False, *, dir_fd=None)** — создает символическую ссылку на объект;
- **os.urandom(n)** — **n** случайных байт. Возможно использование этой функции в криптографических целях;
- **os.path** — модуль, реализующий некоторые полезные функции для работы с путями.

Модуль SYS

- ***Что можете сказать про модуль SYS?***

С помощью модуля **SYS** мы можем получить доступ к переменным, которые взаимодействуют с интерпретатором Python.

```
import sys
# Список аргументов запуска скрипта,
# первым аргументом является полный путь к файлу скрипта
print('sys.argv = ', sys.argv)
# Список путей для поиска модулей
print('sys.path = ', sys.path)
# Полный путь к интерпретатору
print('sys.executable = ', sys.executable)
# Словарь имен загруженных модулей
print('sys.modules = ', sys.modules)
# Информация об операционной системе
print('sys.platform = ', sys.platform)
while True:
    key = input("press 'q' to Exit")
    if key == 'q':
        sys.exit()
        # Вызов данной функции мгновенно завершает работу модуля (скрипта)
```

Запуск скрипта с параметрами

- ***Как запустить скрипт с параметрами?***

Скрипт можно запустить с дополнительными параметрами.

К примеру:

```
> python my_script.py param1 param2 param3
```

С помощью модуля **SYS** можно извлечь эти параметры:

```
import sys
print('sys.argv = ', sys.argv)
```

Получим:

```
sys.argv = [my_script.py', 'param1', 'param2', 'param3']
```

Практическое задание

1. Написать программу, которая будет содержать функцию для получения имени файла из полного пути до него. При вызове функции в качестве аргумента должно передаваться имя файла с расширением. В функции необходимо реализовать поиск полного пути по имени файла, а затем «выделение» из этого пути имени файла (без расширения).

2. Написать программу, которая запрашивает у пользователя ввод числа. На введенное число она отвечает сообщением, целое оно или дробное. Если дробное — необходимо далее выполнить сравнение чисел до и после запятой. Если они совпадают, программа должна возвращать значение **True**, иначе **False**.

3. Создать два списка с различным количеством элементов. В первом должны быть записаны ключи, во втором — значения. Необходимо написать функцию, создающую из данных ключей и значений словарь. Если ключу не хватает значения, в словаре для него должно сохраняться значение **None**. Значения, которым не хватило ключей, необходимо отбросить.

4. Написать программу, в которой реализовать две функции. В первой должен создаваться простой текстовый файл. Если файл с таким именем уже существует, выводим соответствующее сообщение. Необходимо открыть файл и подготовить два списка: с текстовой и числовой информацией. Для создания списков использовать генераторы. Применить к спискам функцию **zip()**. Результат выполнения этой функции должен должен быть обработан и записан в файл таким образом, чтобы каждая строка файла содержала текстовое и числовое значение. Вызвать вторую функцию. В нее должна передаваться ссылка на созданный файл. Во второй функции необходимо реализовать открытие файла и простой

построчный вывод содержимого. Вся программа должна запускаться по вызову первой функции.

5. Усовершенствовать первую функцию из предыдущего примера. Необходимо во втором списке часть строковых значений заменить на значения типа **example345** (строка+число). Далее — усовершенствовать вторую функцию из предыдущего примера (функцию извлечения данных). Дополнительно реализовать поиск определенных подстрок в файле по следующим условиям: вывод первого вхождения, вывод всех вхождений. Реализовать замену всех найденных подстрок на новое значение и вывод всех подстрок, состоящих из букв и цифр и имеющих пробелы только в начале и конце — например, **example345**.

Дополнительные материалы

1. https://pythonworld.ru/moduli/modul-os.html.
2. https://pythonworld.ru/osnovy/with-as-menedzhery-konteksta.html.
3. https://metanit.com/python/tutorial/5.3.php.
4. https://o7planning.org/ru/11417/inheritance-and-polymorphism-in-python.
5. https://metanit.com/python/tutorial/7.4.php.
6. https://pythonworld.ru/osnovy/inkapsulyaciya-nasledovanie-polimorfizm.html.

Используемая литература

1. Учим Python качественно (habr).
2. Самоучитель по Python.
3. Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание).
4. 15 основных вопросов для Python собеседования.
5. 20 вопросов и ответов из интервью на позицию Python-разработчика.