Основы Python

Урок 7. Работа с файловой системой. Исключения в Python



# На этом уроке

1. Научимся работать с содержимым папок и создавать внутри них новые файлы и папки.
2. Познакомимся с обработкой исключений в Python.

## 

## Оглавление

[Работа с файловой системой: модуль os](#_hwud9og5m975)

[Поиск файлов в папке](#_kb1gjzjk3wa6)

[Полезные функции модуля os.path](#_r9awatk1s6jp)

[Модуль os: создание, переименование и удаление папок](#_49oqrjfdomzi)

[Работа с файловой системой: модуль shutil](#_38bkviqsi4j9)

[Функция copyfileobj()](#_clc7w2wij1jd)

[Функция copyfile()](#_jn48ixi0xwd0)

[Функции copy() и copy2()](#_o8qhpx48p56e)

[Работа с файловой системой: рекурсивный обход папок](#_3zte3vmjh9ig)

[Обработка исключительных ситуаций в Python](#_epwqhqivgfnl)

[Блок try...except](#_jpgtwglq8c8m)

[Встроенные классы-исключения](#_9r9izvd03oh3)

[Блок else](#_gc38nby42o6b)

[Блок finally](#_cy3yk44dzump)

[Часто используемые исключения](#_wm411p5efpvi)

[Ключевое слово raise](#_nzl5r1mz634g)

[Практическое задание](#_ph6z0klx9y2w)

[Дополнительные материалы](#_2xcytpi)

[Используемая литература](#_1ci93xb)

# 

# 

# Работа с файловой системой: модуль os

## Поиск файлов в папке

Начнём с простой задачи: найти все файлы с расширением .py в заданной папке:

| import os  folder = r'C:\Python3.8\Lib\site-packages\aiohttp' py\_files = [item  for item in os.listdir(folder)  if item.lower().endswith('.py')] print(py\_files) *# ['abc.py', 'client.py', 'client\_exceptions.py', 'client\_proto.py', ...* |
| --- |

Код простой, но в нём есть нюансы. Префикс “r” перед строкой позволил нам не экранировать обратный слеш — теперь строка воспринимается «сырой» (raw). Имейте в виду, что в таком режиме управляющие символы типа “\n” не будут работать.

**Примечание:** библиотека [aiohttp](https://docs.aiohttp.org/en/stable/) очень часто используется в проектах — рекомендуем установить её в систему и поработать самостоятельно в будущем.

Как вы уже догадались, список файлов в папке мы получили при помощи функции [listdir()](https://docs.python.org/3.8/library/os.html?highlight=listdir#os.listdir) из модуля os. В качестве аргумента передаем ей относительный или абсолютный путь к папке. На выходе получаем список строк: имена всех файлов в папке, кроме “.” и “..”. Дальше работали с этим списком как обычно — взяли только строки, заканчивающиеся интересующим расширением.

Важная особенность функции listdir() — она «видит» только сущности в текущей папке, не заглядывая в подпапки.

Вторая особенность — мы получаем только имена сущностей, а не пути к ним. Придется воссоздавать путь при помощи функции [join()](https://docs.python.org/3.8/library/os.path.html#os.path.join) из модуля [os.path](https://docs.python.org/3.8/library/os.path.html#module-os.path):

| ... py\_files = [os.path.join(folder,item)  for item in os.listdir(folder)  if item.lower().endswith('.py')] print(py\_files) *# ['C:\\Python3.8\\Lib\\site-packages\\aiohttp\\abc.py', ...* |
| --- |

Вы заметили, как Python экранирует обратные слеши при выводе через функцию print()?

Третья особенность — мы не можем сразу узнать параметры объекта файловой системы: его тип (файл или папка), объём, время изменения или создания и т.д. Если в алгоритме понадобится эта информация, придётся использовать дополнительные функции из модуля [os.path](https://docs.python.org/3.8/library/os.path.html#module-os.path): [isfile()](https://docs.python.org/3.8/library/os.path.html#os.path.isfile), [isdir()](https://docs.python.org/3.8/library/os.path.html#os.path.isdir) или из os: [stat()](https://docs.python.org/3.8/library/os.html#os.stat).

Например, покажем все папки в корне фреймворка [Django](https://www.djangoproject.com/):

| folder = r'C:\Python3.8\Lib\site-packages\django' django\_dirs = [item  for item in os.listdir(folder)  if os.path.isdir(os.path.join(folder, item))] print(django\_dirs) *# ['apps', 'bin', 'conf', 'contrib', 'core', 'db', 'dispatch', 'forms',* *# 'http', 'middleware', 'template', 'templatetags', 'test', 'urls', 'utils',* *# 'views', '\_\_pycache\_\_']* |
| --- |

**Примечание:** для запуска некоторых скриптов в этом уроке придется [установить](https://docs.djangoproject.com/en/3.1/topics/install/#installing-an-official-release-with-pip) фреймворк Django. Его размещение в вашей системе может отличаться от того, которое указано в примерах. Рекомендуем использовать виртуальное окружение.

Этот код работает, но для каждой сущности вызывается функция isdir() — это не очень хорошо и может привести к существенной потере скорости. Давайте убедимся в этом.

Создадим в корне урока папку some\_data, а в ней 1000 небольших файлов разного размера. Можно сделать это при помощи скрипта:

| import os import random  folder = 'some\_data' letters = [chr(code) for code in range(ord('a'), ord('z') + 1)] for \_ in range(10 \*\* 3):  f\_name = ''.join(random.sample(letters, random.randint(5, 10)))  f\_content = bytes(random.randint(0, 255)   for \_ in range(random.randrange(10 \*\* 5)))  with open(os.path.join(folder, f'{f\_name}.bin'), 'wb') as f:  f.write(f\_content) |
| --- |

Теперь решим задачу: вывести на экран имена и размер файлов размером не более 15 КБ. Решать будем двумя способами: при помощи функции listdir() и при помощи альтернативной функции [scandir()](https://docs.python.org/3.8/library/os.html?highlight=listdir#os.scandir), которая возвращает генератор объектов [DirEntry](https://docs.python.org/3.8/library/os.html?highlight=listdir#os.DirEntry). Это позволяет избежать дополнительных [системных вызовов](https://en.wikipedia.org/wiki/Stat_(system_call)) и существенно повысить скорость.

| import os from time import perf\_counter  folder = 'some\_data' start = perf\_counter() size\_threshold = 15 \* 2 \*\* 10 small\_files = [item  for item in os.listdir(folder)  if os.stat(os.path.join(folder, item)).st\_size < size\_threshold] print(len(small\_files), perf\_counter() - start) *# 155 2.271335837* start = perf\_counter() small\_files\_2 = [item.name  for item in os.scandir(folder)  if item.stat().st\_size < size\_threshold] print(len(small\_files\_2), perf\_counter() - start) print(small\_files == small\_files\_2) *# 155 0.0739816499999999* *# True* |
| --- |

В Windows получили ускорение в 30 раз. Теперь решим первую задачу — просто получим список файлов в папке:

| import os from time import perf\_counter  folder = 'some\_data' start = perf\_counter() all\_files = [item  for item in os.listdir(folder)] print(len(all\_files), perf\_counter() - start) *# 1000 0.056053622000000025* start = perf\_counter() all\_files\_2 = [item.name  for item in os.scandir(folder)] print(len(all\_files\_2), perf\_counter() - start) print(all\_files == all\_files\_2) *# 1000 0.04480954299999995* *# True* |
| --- |

Как и следовало ожидать, разницы нет.

При работе с функцией scandir() нужно всегда помнить, что она возвращает итератор, а значит, «захватывает» определённые ресурсы до тех пор, пока он не закрыт. Поэтому если вы используете его вне другого итератора/генератора или comprehension, необходимо использовать менеджер контекста, который будет эти ресурсы «освобождать», или в коде вызывать метод [.close()](https://docs.python.org/3.8/library/os.html?highlight=listdir#os.scandir.close) (в этой ссылке есть пример).

## Полезные функции модуля os.path

В модуле os.path есть ещё много полезных функций для работы с файловой системой:

* [abspath()](https://docs.python.org/3.8/library/os.path.html#os.path.abspath) — возвращает абсолютный путь;
* [basename()](https://docs.python.org/3.8/library/os.path.html#os.path.basename) — возвращает имя файла из абсолютного или относительного пути;
* [dirname()](https://docs.python.org/3.8/library/os.path.html#os.path.dirname) — возвращает имя (путь) папки, в которой расположен файл;
* [split()](https://docs.python.org/3.8/library/os.path.html#os.path.split) — делит путь к файлу на путь к папке и имя файла (заменяет вызов двух предыдущих функций);
* [relpath()](https://docs.python.org/3.8/library/os.path.html#os.path.relpath) — определяет путь к файлу относительно другой папки, не обращается к реальной файловой системе, чисто вычисления (полезна при сохранении путей к файлам в базе данных относительно заданного корня);
* [join()](https://docs.python.org/3.8/library/os.path.html#os.path.join) — склеивает путь из частей (надеемся, вы не делаете это через строки!);
* [exists()](https://docs.python.org/3.8/library/os.path.html#os.path.exists) — проверяет существование объекта файловой системы.

Закрепим в скриптах:

| import os  root = r'C:\Python3.8\Lib\site-packages\django' folder = r'C:\Python3.8\Lib\site-packages\django\contrib\admin' django\_admin\_dirs = [  os.path.relpath(item, root)  for item in os.scandir(folder)  if item.is\_dir() and not item.name.startswith('\_') ] print(django\_admin\_dirs) *# ['contrib\\admin\\locale', 'contrib\\admin\\migrations',*  *# 'contrib\\admin\\static', 'contrib\\admin\\templates',* *# 'contrib\\admin\\templatetags', 'contrib\\admin\\views']* |
| --- |

Получили пути к папкам админки Django относительно корня — размещения проекта в системе (для Windows, в других системах будет другой путь). Еще пример:

| import os  curr\_file = r'C:\Python3.8\Lib\site-packages\django\http\request.py' print('exists', os.path.exists(curr\_file)) *# exists ok True* f\_dir, f\_name = os.path.split(curr\_file) print(f\_dir, f\_name, sep=' | ') *# C:\Python3.8\Lib\site-packages\django\http | request.py* print('dirname ok', f\_dir == os.path.dirname(curr\_file)) *# dirname ok True* print('basename ok', f\_name == os.path.basename(curr\_file)) *# basename ok True* print('abspath ok', curr\_file == os.path.abspath(curr\_file)) *# abspath ok True* curr\_file\_rel = os.path.relpath(curr\_file, root) print(curr\_file\_rel) *#http\request.py* print('relpath ok', curr\_file == os.path.join(root, curr\_file\_rel)) *# relpath ok True* |
| --- |

Важно помнить, что функция split() из модуля os.path всегда возвращает список из двух элементов, в отличие от метода .split() класса str, который может вернуть список любого размера.

## Модуль os: создание, переименование и удаление папок

Создадим программно папку sample\_dir:

| import os  dir\_name = 'sample\_dir' if not os.path.exists(dir\_name):  os.mkdir(dir\_name) |
| --- |

Использовали функцию [mkdir()](https://docs.python.org/3.8/library/os.html#os.mkdir) с интуитивно понятным синтаксисом. Как вы думаете, зачем мы добавили проверку отсутствия папки? Верно: чтобы не было ошибки, если эта папка уже была создана ранее.

А что, если необходимо создать иерархию папок? Например, в проекте нужна папка data, а в ней — папка src:

| import os  dir\_path = os.path.join('data', 'src') if not os.path.exists(dir\_path):  os.mkdir(dir\_path) *# ...FileNotFoundError:...* |
| --- |

Кому-то ошибка покажется неожиданной, но мы сразу пытаемся создать больше одной папки: родительскую, и в ней — ещё одну. В таком случае необходимо использовать функцию [makedirs()](https://docs.python.org/3.8/library/os.html#os.makedirs):

| import os  dir\_path = os.path.join('data', 'src') if not os.path.exists(dir\_path):  os.makedirs(dir\_path) |
| --- |

При работе с модулем os необходимо быть очень внимательными — много нюансов.

Создадим вручную папку first\_dir и наполним её любым содержимым. Попробуем её переименовать:

| import os  dir\_name = 'first\_dir' new\_dir\_name = 'first\_new\_dir' if os.path.exists(dir\_name) and not os.path.exists(new\_dir\_name):  os.rename(dir\_name, new\_dir\_name) |
| --- |

Функция [rename()](https://docs.python.org/3.8/library/os.html#os.rename) позволяет переименовывать папки и файлы: первый аргумент — исходное имя (путь), второй — новое имя (путь). Обратите внимание, что логика стала сложнее: теперь проверяем наличие исходной папки и отсутствие папки с новым именем. Скоро мы узнаем, как можно упростить этот момент. Меняется ли поведение скрипта при нескольких запусках? Совпало ли оно с тем, что вы предполагали?

Попробуйте повторить эксперимент для имени:

| ... new\_dir\_name = '../first\_out\_dir' ... |
| --- |

Всё ли получилось при запуске? Да, но папка не просто переименовалась, а оказалась в папке на уровень выше в файловой системе (родительская папка для корня урока).

Что **важное** узнали из этого эксперимента с функцией rename():

* можем переименовывать пустые и непустые папки (если хватает прав доступа);
* можем перемещать их в новую локацию вместе с содержимым (как если бы сделали «вырезать» и «вставить» в операционной системе).

Теперь создадим вручную папку second\_dir и попробуем её удалить при помощи функции [remove()](https://docs.python.org/3.8/library/os.html#os.remove):

| import os  to\_remove\_dir\_name = 'second\_dir' if os.path.exists(to\_remove\_dir\_name):  os.remove(to\_remove\_dir\_name) *# Windows:...PermissionError: [WinError 5] Отказано в доступе: 'second\_dir' # Linux:...IsADirectoryError: [Errno 21] Is a directory: 'second\_dir'* |
| --- |

Поймали ошибку, ведь эта функция предназначена для удаления файлов, а не папок. Попробуйте выполнить этот код для файла — всё должно получиться. Для удаления папок в Python необходимо использовать функцию [rmdir()](https://docs.python.org/3.8/library/os.html#os.rmdir):

| import os  to\_remove\_dir\_name = 'second\_dir' if os.path.exists(to\_remove\_dir\_name):  os.rmdir(to\_remove\_dir\_name) *# Windows:...OSError: [WinError 145] Папка не пуста: 'second\_dir'* *# Linux:...OSError: [Errno 39] Directory not empty: 'second\_dir'* |
| --- |

Опять ошибка — папка не пустая.

Создадим пустую папку empty\_dir — теперь удаление будет выполнено успешно.

# Работа с файловой системой: модуль shutil

Надеемся, теперь вы понимаете, почему для удаления папок обычно используют функцию [rmtree()](https://docs.python.org/3.8/library/shutil.html#shutil.rmtree) из модуля [shutil](https://docs.python.org/3.8/library/shutil.html):

| import os import shutil  to\_remove\_dir\_name = 'second\_dir' if os.path.exists(to\_remove\_dir\_name):  shutil.rmtree(to\_remove\_dir\_name) |
| --- |

Разумеется, файл при помощи этой функции удалить не получится — для этих целей, как мы уже знаем, есть функция remove() в модуле os.

Что ещё полезного есть в модуле shutil? Функции для копирования файлов:

* [copyfileobj()](https://docs.python.org/3.8/library/shutil.html#shutil.copyfileobj) — копирование одного файлового объекта в другой;
* [copyfile()](https://docs.python.org/3.8/library/shutil.html#shutil.copyfile) — копирование содержимого одного файла в другой (настройки доступа не копируются);
* [copy()](https://docs.python.org/3.8/library/shutil.html#shutil.copy) — копирование файла (копируются настройки доступа);
* [copy2()](https://docs.python.org/3.8/library/shutil.html#shutil.copy2) -— копирование файла (копируются настройки доступа и метаданные — о них подробнее позже).

Тут начинается непростой материал. Для вас это будет своего рода тест на абстрактное мышление и умение различать детали. Важно понимать: «The shutil module offers a number of high-level operations…» — модуль shutil реализован на высоком уровне (уровне самого интерпретатора Python). Это значит, что скорость работы не всегда будет высокой.

## Функция copyfileobj()

Начнем с функции copyfileobj(). Самый просто способ понять, что она делает, — посмотреть исходники:

shutil.py

| ... def copyfileobj(fsrc, fdst, length=16\*1024):  """copy data from file-like object fsrc to file-like object fdst"""  while 1:  buf = fsrc.read(length)  if not buf:  break  fdst.write(buf) ... |
| --- |

Важно понять, что аргументы функции — файловые объекты (мы уже работали с ними на предыдущем уроке). Видим, что функция читает данные порциями длиной length из источника и записывает их в файл-получатель. Первое: это значит, что данные загружаются в оперативную память, если сделать значение length большим, можем её переполнить. Второе: если сделать значение length маленьким, получим большое количество обращений к диску и существенное замедление работы скрипта. Третье: если из файлового объекта-источника была прочитана порция данных до вызова функции copyfileobj(), в файл-получатель эта порция не попадёт — чтение будет происходить с текущего положения курсора. Четвёртое: ответственность за закрытие файловых объектов лежит на нас — необходимо использовать менеджер контекста или вызывать метод .close() после вызова функции copyfileobj().

Практический пример. Создадим файл:

data/hello.txt

| Привет всем, добравшимся до 7-го урока. Копируем файлы. |
| --- |

Выполним скрипт:

| import random import shutil  for \_ in range(3):  with open('data/hello.txt', encoding='utf-8') as src:  with open('data/summary.txt', 'a', encoding='utf-8') as dst:  head\_size = random.randrange(21)  print(head\_size, src.read(head\_size))  shutil.copyfileobj(src, dst) *# 18 Привет всем, добра* *# 14 Привет всем, д* *# 4 Прив* |
| --- |

В результате получим новый текстовый файл:

data/summary.txt

| вшимся до 7-го урока. Копируем файлы.обравшимся до 7-го урока. Копируем файлы.ет всем, добравшимся до 7-го урока. Копируем файлы. |
| --- |

Трижды читали исходный файл — каждый раз считывали случайное число символов и оставшееся содержимое копировали в новый текстовый файл. Функция copyfileobj()полезна в тех случаях, когда мы работаем с файловыми объектами, а не с именами файлов. В официальной документации есть [пример](https://docs.python.org/3/howto/urllib2.html#fetching-urls), где читаются данные по URL-адресу и сохраняются во временный файл.

**\***Также функция copyfileobj() будет полезна при организации конвейеров обработки данных — можно получить информацию от предыдущего обработчика через атрибут .stdout объекта [subprocess.Popen](https://docs.python.org/3/library/subprocess.html#subprocess.Popen) и сохранить в файл.

## Функция copyfile()

По сути, является обёрткой над copyfileobj(), работающей с именами файлов и с дополнительными проверками. Настоятельно рекомендуем посмотреть исходный код этой функции в файле shutil.py и ответить на вопросы: проверяется ли совпадение источника и назначения (да)? Может ли назначение быть папкой (нет)? Проверяется ли специальный тип источника: сокет, именованный канал (да)? Обрабатываются ли особым образом [символические ссылки](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D1%81%D1%8B%D0%BB%D0%BA%D0%B0) (да)?

Если всё хорошо и в качестве источника задан путь к обычному файлу, внутри менеджеров контекста вызывается функция copyfileobj().

Итак, главное отличие copyfile() от copyfileobj() — аргументами должны быть пути к файлам, а не файловые объекты. Общее у этих функций то, что ни метаданные, ни настройки доступа исходного файла не копируются.

## Функции copy() и copy2()

Работают практически одинаково: вместе с файлом копируются настройки доступа к нему. Функция copy2() также копирует метаданные файла: дату создания, изменения, последнего доступа и т.д. Важная особенность этих функций — адресом назначения может быть не только путь к файлу, но и путь к папке назначения: при этом сохранится оригинальное имя файла (похоже на копирование файла в терминале операционной системы).

Если посмотреть исходники этих функций, увидим, что обе они используют функцию copyfile(). Только в copy() после нее вызывается функция [copymode()](https://docs.python.org/3.8/library/shutil.html#shutil.copymode), а в copy2() — функция [copystat()](https://docs.python.org/3.8/library/shutil.html#shutil.copystat).

Проведём эксперимент (желательно в nix-системе). Создадим папку new\_data. Зададим для созданного ранее файла data/summary.txt разрешения командой (если вы работаете в Windows, этот шаг делать не нужно):

| chmod 777 data/summary.txt |
| --- |

Теперь запустим скрипт:

| import os from shutil import copyfile, copy, copy2   def show\_stat(f\_path):  stat = os.stat(f\_path)  print('{f\_p}:\n\tperm - {perm}, modify {m\_t:.0f}, access {a\_t:.0f}'.format(  f\_p=f\_path,  perm=oct(stat.st\_mode),  m\_t=stat.st\_mtime,  a\_t=stat.st\_atime,  ))   src = 'data/summary.txt' show\_stat(src) show\_stat(copyfile(src, 'new\_data/summary\_clone.txt')) show\_stat(copy(src, 'new\_data')) show\_stat(copy2(src, 'new\_data/summary\_clone\_2.txt')) *# data/summary.txt:* *# perm - 0o100777, modify 1610993259, access 1611080145* *# new\_data/summary\_clone.txt:* *# perm - 0o100664, modify 1611080387, access 1611080387* *# new\_data/summary.txt:* *# perm - 0o100777, modify 1611080387, access 1611080387* *# new\_data/summary\_clone\_2.txt:* *# perm - 0o100777, modify 1610993259, access 1611080145* |
| --- |

Видим, что всё подтвердилось:

* copyfile() — не скопировала настройки доступа;
* copy() — сработала с аргументом в виде имени папки, скопировала настройки доступа, но не скопировала метаданные;
* copy2() — скопировала всё.

Если вы запустите скрипт в Windows, то разницу в настройках доступа не обнаружите — особенности операционной системы.

**Замечание:** все функции, кроме copyfileobj(), возвращают путь к скопированному файлу.

**Важно:** все функции являются высокоуровневыми — работают на уровне интерпретатора, в некоторых случаях эффективнее будет использовать средства операционной системы (примеры можно посмотреть [здесь](https://stackoverflow.com/questions/123198/how-do-i-copy-a-file-in-python)). Начиная с Python 3.8, интерпретатор сам пытается использовать [наиболее эффективные](https://docs.python.org/3/library/shutil.html#platform-dependent-efficient-copy-operations) для конкретной операционной системы способы копирования.

# Работа с файловой системой: рекурсивный обход папок

Мы уже умеем работать с содержимым конкретной папки при помощи функций listdir() и scandir() модуля os (помните разницу между ними?). А если необходимо заходить в папки и подпапки? Конечно, можно написать скрипт с использованием известных функций, но есть более эффективный способ — функция [walk()](https://docs.python.org/3/library/os.html#os.walk) модуля os. Для примера просканируем уже знакомую нам папку с фреймворком django и создадим словарь, в котором ключами будут расширения файлов, а значениями — пути к этим файлам относительно корня проекта:

| import os from collections import defaultdict from os.path import relpath  import django  root\_dir = django.\_\_path\_\_[0] django\_files = defaultdict(list) for root, dirs, files in os.walk(root\_dir):  for file in files:  ext = file.rsplit('.', maxsplit=1)[-1].lower()  rel\_path = relpath(os.path.join(root, file), root\_dir)  django\_files[ext].append(rel\_path)  for ext, files in sorted(django\_files.items(),  key=lambda x: len(x[1]), reverse=True):  print(f'{ext}: {len(files)}')  print('\nPY FILES') print(\*sorted(django\_files['py'])[:10], sep='\n') |
| --- |

Возможный результат работы скрипта:

| mo: 1125 po: 1125 py: 838 pyc: 838 html: 121 js: 78 svg: 26 css: 14 py-tpl: 12 txt: 8 woff: 3 license: 3 md: 2 kml: 2 xml: 2 gz: 1  PY FILES \_\_init\_\_.py \_\_main\_\_.py apps\\_\_init\_\_.py apps\config.py apps\registry.py bin\django-admin.py conf\\_\_init\_\_.py conf\global\_settings.py conf\locale\\_\_init\_\_.py conf\locale\ar\\_\_init\_\_.py |
| --- |

Разберём особенности кода.

Первое — видим цикл-итератор с тремя аргументами: root, dirs, files. Непривычно, но на самом деле все просто: root — это текущая папка, а dirs и files — папки и файлы, которые в ней есть. Можете запустить простой скрипт, чтобы в этом убедиться:

| import os  import django  root\_dir = django.\_\_path\_\_[0] for root, dirs, files in os.walk(root\_dir):  print(root, len(dirs), len(files)) |
| --- |

Второе — получили корневую папку пакета при помощи [особого](https://docs.python.org/3.8/reference/import.html?highlight=__path__#__path__) атрибута \_\_path\_\_. Теперь код будет работать «из коробки» — не надо вручную прописывать путь. Можете пока воспринимать этот трюк как магию.

Третье — использовали функцию [defaultdict()](https://docs.python.org/3.8/library/collections.html?highlight=defaultdict#collections.defaultdict) из модуля collections: код стал сильно проще. Сравните с вариантом, где используется обычный словарь:

| ... django\_files = {} for root, dirs, files in os.walk(root\_dir):  for file in files:  ext = file.rsplit('.', maxsplit=1)[-1].lower()  rel\_path = relpath(os.path.join(root, file), root\_dir)  if ext not in django\_files:  django\_files[ext] = []  django\_files[ext].append(rel\_path) ... |
| --- |

Нам приходится проверять наличие ключа в словаре, и, если его нет, создавать пустой список для этого ключа. Согласитесь, что вариант defaultdict(list) значительно проще. Подумайте, как работал бы код: defaultdict(dict)? Верно: создавал бы пустой словарь для каждого нового ключа.

Остальная часть кода вам должна быть понятна. Если это не так, нужно повторить предыдущий материал.

**Замечание:** начиная с версии Python 3.5, функция walk() использует os.scandir() вместо os.listdir() — мы уже знаем, что это приводит к существенному росту скорости в некоторых случаях.

# Обработка исключительных ситуаций в Python

## Блок try...except

Уже многократно в курсе мы сталкивались с ситуациями, когда при выполнении скрипта могли возникать ошибки: избегали их при помощи дополнительных проверок. На самом деле это не совсем правильно — в python-разработке приветствуется стиль [ask for forgiveness than permission](https://docs.python.org/3/glossary.html?highlight=ask%20forgiveness%20permission#term-eafp). То есть мы не просим разрешения и не делаем проверок — пробуем выполнить код и имеем дело с последствиями. Для этого используется конструкция [try...except](https://docs.python.org/3/tutorial/errors.html#handling-exceptions):

| f\_path = 'new\_one.txt' try:  with open(f\_path, 'r', encoding='utf-8') as f:  content = f.read()  print(content) except (FileNotFoundError, EOFError) as e:  print(f'concrete error: {e}') except Exception as e:  print(f'global error: {e}') *# concrete error: [Errno 2] No such file or directory: 'new\_one.txt'* |
| --- |

Между ключевыми словами try и except располагаем код, где могут быть ошибки. **Важно:** чем меньшая часть кода будет в этом участке, тем проще искать и обрабатывать ошибку.

После ключевого слова except пишем класс перехватываемого исключения или кортеж из нескольких классов (**обязательно** в круглых скобках!), если мы хотим сделать один обработчик для них.

Как правило, после класса исключения дописываем as e — получаем в переменной e объект ошибки.

Хорошей практикой является логирование ошибки или вывод ее в консоль — не стоит «замалчивать», иначе в будущем потратите **очень** много времени на правку багов.

## Встроенные классы-исключения

Обязательно изучите иерархию [встроенных классов-исключений](https://docs.python.org/3.8/library/exceptions.html#exception-hierarchy). Её фрагмент:

| BaseException  +-- SystemExit  +-- KeyboardInterrupt  +-- Exception  +-- StopIteration  +-- ArithmeticError  | +-- FloatingPointError  | +-- OverflowError  | +-- ZeroDivisionError  +-- AssertionError  +-- AttributeError  +-- EOFError  +-- ImportError  +-- LookupError  | +-- IndexError  | +-- KeyError  +-- MemoryError  +-- NameError  +-- OSError  | +-- FileExistsError  | +-- FileNotFoundError  | +-- InterruptedError  | +-- IsADirectoryError  | +-- NotADirectoryError  | +-- PermissionError  | +-- TimeoutError  +-- RuntimeError  +-- SyntaxError  +-- SystemError  +-- TypeError  +-- ValueError  +-- Warning |
| --- |

Официально мы ещё не обсуждали понятие «[наследование](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))», но при разговоре об исключениях будем его использовать. Если по-простому: класс-наследник получает все атрибуты и методы родителя. Ещё один момент, связанный с наследованием, — возможность работать на разных уровнях иерархии. Это как в случае с папками на диске. Можем копировать отдельные файлы (наследников папки), а можем всю папку (родителя). Только при обработке исключений не копируем файлы, а перехватываем ошибки определенных типов.

Исключения могут быть перехвачены в блоке try...except: обычно здесь ловим исключения, унаследованные от класса Exception (в редких случаях от BaseException, например, KeyboardInterrupt и SystemExit, возникающие вне исполняемого кода). По сути можем сделать единственный обработчик с классом Exception — он поймает любую ошибку. Но это будет грубо и используется только в очень простых случаях.

Другая крайность: ловить конкретные исключения, как мы поступили с FileNotFoundError.

Во многих случаях достаточно перехватывать чуть более глобальные классы исключений:

* OSError — что-то не так при взаимодействии с операционной системой (не конкретно);
* ArithmeticError — проблема с арифметическими операциями (без деталей: ошибка деления или что-то ещё).

То есть если написать в цепочке обработчиков:

| ... except OSError as e:  print(f'OS says: {e}') ... |
| --- |

Получим код, обрабатывающий любые исключения, которые стоят ниже в иерархии: и отсутствие файла FileNotFoundError при попытке доступа к нему, и существование файла FileExistsError при попытке его перезаписать, и попытку доступа к папке IsADirectoryError вместо файла и т.д.

А если запишем:

| ... except FileNotFoundError as e:  print(f'no file: {e.filename}') ... |
| --- |

Будем обрабатывать только ситуацию, когда файл не был найден. Остальные ошибки будут исправляться в других обработчиках или вообще могут остаться без обработки (такое бывает).

**Замечание:** откуда мы узнали, что у объекта ошибки есть атрибут filename? При помощи вызова dir(e).

У каждого исключения есть свои особенности. Когда мы работаем с конкретным классом, можем позволить обращение к атрибутам, которые есть именно у этого класса (как мы здесь и сделали). Если бы ловили в этом примере более глобальный класс (например, Exception), могли бы получить объект ошибки любого из классов-наследников. Это значит, что обработчик придется писать более абстрактным — вместо “e.filename” уже будет просто “e” (как и делают в большинстве случаев).

**Замечание:** иногда в обработчиках ошибок обращаются к атрибуту .args объекта ошибки. Попробуйте сами посмотреть разницу и внимательно изучите [примеры](https://docs.python.org/3/tutorial/errors.html#handling-exceptions).

**Очень важно** почувствовать, какие возможности даёт вам такой подход к работе с исключениями. Как zoom-объектив у фотоаппарата: захотели — взяли крупнее, захотели — мельче. Можно (и нужно!) создавать свои иерархии исключений для проектов, но это уже задача другого курса.

**Нюанс:** после перехвата исключения в конкретном блоке except его дальнейшая обработка прекращается, остальным обработчикам ничего не «достаётся». Поэтому **очень важен** порядок, в котором мы «ловим» исключения — **от частных к общим**.

Вопрос: можно ли писать блок except с классом Exception до других обработчиков? Правильно: **нельзя**! Почему? Как мы уже говорили, это самое глобальное исключение и оно перехватывает **любую** ошибку. Давайте попробуем:

| f\_path = 'new\_one.txt' try:  with open(f\_path, 'r', encoding='utf-8') as f:  content = f.read()  print(content) except Exception as e:  print(f'global error: {e}') except (FileNotFoundError, EOFError) as e:  print(f'concrete error: {e}') *# global error: [Errno 2] No such file or directory: 'new\_one.txt'* |
| --- |

Какой обработчик сработал теперь? Самый первый. И он будет срабатывать всегда, даже если будут ошибки FileNotFoundError или EOFError — их поймает первый обработчик. Наш код фактически эквивалентен:

| f\_path = 'new\_one.txt' try:  with open(f\_path, 'r', encoding='utf-8') as f:  content = f.read()  print(content) except Exception as e:  print(f'global error: {e}') |
| --- |

## Блок else

Иногда необходимо выполнить некоторые манипуляции в случае, если ошибок не было. Для этого можно написать блок else:

| f\_path = \_\_file\_\_ try:  with open(f\_path, 'r', encoding='utf-8') as f:  content = f.read() except Exception as e:  print(f'global error: {e}') else:  print(content) |
| --- |

Код в блоке else выполняется **только** если ни одно исключение не было перехвачено — всё прошло хорошо. Переменная [\_\_file\_\_](https://docs.python.org/3/reference/import.html#__file__) хранит имя файла, поэтому код всегда будет выполняться без ошибок и выводить сам себя на экран. Если написать путь к несуществующему файлу, случится ошибка, и код внутри блока else не выполнится.

Давайте более вдумчиво сравним этот пример с предыдущим. На самом деле мы раньше писали код print(content) внутри блока try, и он тоже выполнялся только если ошибки чтения файла не было. В чём разница? Первое: мы уменьшили количество кода в блоке try — это очень хорошо (упрощается исправление багов в будущем). Если какая-то ошибка случится в блоке else, она будет обрабатываться в другой части кода. Второе: мы можем принудительно вызвать исключение в блоке else (зачем это нужно, скажем чуть позже).

## Блок finally

Бывают ситуации, когда надо выполнить некоторый код независимо от того, была ошибка или нет. Рассмотрим пример: необходимо вести лог математических вычислений — деления двух чисел. Можно написать такой код:

| f\_path = 'calc\_log.txt' f = open(f\_path, 'a', encoding='utf-8') try:  x = float(input('enter x val: '))  y = float(input('enter y val: ')) except ValueError as e:  print(f'wrong val: {e}') else:  result = x / y  f.write(f'{x} / {y} = {result}\n') f.close() |
| --- |

Здесь мы считаем, что ошибки, связанные с работой с файлами, обрабатываются где-то на другом уровне проекта, поэтому открытие файла происходит не в блоке try. Пусть ошибка деления на ноль тоже перехватывается не здесь (это вопрос разделения ответственности фрагментов кода).

Давайте подумаем, что будет, если пользователь введет данные, которые нельзя корректно преобразовать в вещественное число (например, напишет запятую вместо точки)? Сработает исключение [ValueError](https://docs.python.org/3/library/exceptions.html?highlight=valueerror#ValueError) (попытка передать функции аргумент не того **значения** — “3,5” вместо “3.5” для функции float() — или выполнить действие над операндом с не тем **значением**), и мы увидим сообщение об ошибке. Результат не будет вычислен, и записи в файл не произойдет. Открытый в самом начале файл закроется, и скрипт завершит работу без ошибок. Подумайте минуту, что может быть не так? Конечно, ситуация с делением на ноль: вы ввели оба числа (второе число ноль), ошибок не было — выполняется код в блоке else. В нем происходит деление на ноль, что приводит к ошибке [ZeroDivisionError](https://docs.python.org/3/library/exceptions.html?highlight=valueerror#ZeroDivisionError). Выполнение скрипта прекращается. Но в реальном проекте всё может быть сложнее: ошибку перехватывает внешний к этому коду обработчик, и что-то происходит далее. Вопрос: будет ли закрыт файл? Давайте проверим. Для этого обернем код в функцию:

| def do\_calc(f\_path):  f = open(f\_path, 'a', encoding='utf-8')  try:  x = float(input('enter x val: '))  y = float(input('enter y val: '))  except ValueError as e:  print(f'wrong val: {e}')  else:  result = x / y  f.write(f'{x} / {y} = {result}\n')  print('closing file')  f.close()   if \_\_name\_\_ == '\_\_main\_\_':  f\_path = 'calc\_log.txt'  try:  do\_calc(f\_path)  except ZeroDivisionError:  print('fault: Zero division')  except Exception as e:  print(f'global error: {e}') |
| --- |

Пробуем поделить 18 на 2:

| enter x val: 18 enter y val: 2 closing file |
| --- |

Все хорошо. Файл закрыт.

Теперь делим 18 на 0:

| enter x val: 18 enter y val: 0 fault: Zero division |
| --- |

Так как ошибка случилась раньше кода, где мы закрывали файл, выполнение скрипта было прервано и файл так и не был закрыт. И тут на помощь приходит блок [finally](https://docs.python.org/3/reference/compound_stmts.html#finally) — код, написанный в нём, выполняется **всегда**!

**Замечание:** даже если у вас в конце блока try был return, и всё прошло хорошо, перед выходом из функции обязательно выполнится код блока finally (рекомендуем проверить такое поведение самостоятельно). Это **очень** важный нюанс.

Пофиксим баг в нашем коде:

| def do\_calc(f\_path):  f = open(f\_path, 'a', encoding='utf-8')  try:  x = float(input('enter x val: '))  y = float(input('enter y val: '))  except ValueError as e:  print(f'wrong val: {e}')  else:  result = x / y  f.write(f'{x} / {y} = {result}\n')  finally:  print('closing file')  f.close()   if \_\_name\_\_ == '\_\_main\_\_':  f\_path = 'calc\_log.txt'  try:  do\_calc(f\_path)  except ZeroDivisionError:  print('fault: Zero division')  except Exception as e:  print(f'global error: {e}')  *# enter x val: 18* *# enter y val: 0* *# closing file* *# fault: Zero division* |
| --- |

Теперь файл всегда будет закрываться.

Следует признать, что этот пример носит синтетический характер и придуман в учебных целях — показать смысл блока finally. Но если вы не освободите ресурс в более сложном коде (например, если бы после вызова do\_calc() происходило бы что-то еще), могут быть реальные проблемы.

## Часто используемые исключения

Помимо уже встретившихся нам исключений (ValueError, ZeroDivisionError, FileExistsError, FileNotFoundError), достаточно часто в коде бывают полезны:

* [AttributeError](https://docs.python.org/3/library/exceptions.html?highlight=valueerror#AttributeError) — попытка обратиться к несуществующему атрибуту или методу;
* [IndexError](https://docs.python.org/3/library/exceptions.html?highlight=valueerror#IndexError) — ошибка индекса (в списке), обращение к элементу с индексом за пределами существующих;
* [KeyError](https://docs.python.org/3/library/exceptions.html?highlight=valueerror#KeyError) — ошибка ключа (в словаре), обращение к элементу словаря, которого в нём нет;
* [TypeError](https://docs.python.org/3/library/exceptions.html?highlight=valueerror#TypeError) — попытка передать функции аргумент не того **типа** (например, преобразовать список в число float([1,])) или выполнить операцию над операндом не того **типа**.

Примеры кода, вызывающие эти исключения:

| days = (1, 5, 6, 17) print(days[15]) *# ...IndexError: tuple index out of range* |
| --- |

| days = (1, 5, 6, 17) print(days[15]) *# ...IndexError: tuple index out of range* |
| --- |

| week = {'mon': 'пн', 'tue': 'вт'} print(week['wed']) *# ...KeyError: 'wed'* |
| --- |

| days = (1, 5, 6, 17) print(float(days)) *# ...TypeError: float() argument must be a string or a number, not 'tuple'* |
| --- |

| price = '1,7' print(float(price)) *# ...ValueError: could not convert string to float: '1,7'* |
| --- |

Обращаем ваше внимание на два последних примера: вам должна быть понятна **существенная** разница в между ними — неверный тип данных (класс объекта) или неверное значение.

## Ключевое слово raise

Это, пожалуй, самая сложная часть работы с исключениями в Python. Да, иногда мы специально вызываем (или, как еще говорят, «поднимаем», «выбрасываем») исключения в коде. Это может выглядеть как специальное создание ошибок в ходе выполнения программы. Причём своими же руками. Но при более пристальном и глубоком изучении темы должно прийти понимание, насколько это крутая возможность языка. Давайте по порядку.

**Первая причина:** мы хотим, чтобы ошибку продолжил обрабатывать ещё один обработчик. По факту мы просто хотим пробросить её дальше. Вернемся к нашему примеру с вычислениями. Пусть нам необходимо в случае любой ошибки завершить скрипт с некоторым числом, отличным от нуля:

| def do\_calc(f\_path):  f = open(f\_path, 'a', encoding='utf-8')  try:  x = float(input('enter x val: '))  y = float(input('enter y val: '))  except ValueError as e:  print(f'wrong val: {e}')  raise ValueError  else:  result = x / y  f.write(f'{x} / {y} = {result}\n')  finally:  f.close()   if \_\_name\_\_ == '\_\_main\_\_':  f\_path = 'calc\_log.txt'  try:  do\_calc(f\_path)  except ZeroDivisionError:  print('fault: Zero division')  exit(1)  except Exception as e:  print(f'global error: {e}')  exit(2)  *# enter x val: 18 # enter y val: 2,0 # wrong val: could not convert string to float: '2,0' # global error: could not convert string to float: '2,0' #  # Process finished with exit code 2* |
| --- |

Что мы сделали? Обработали исключение внутри функции и пробросили его дальше (вы же помните, что штатное поведение Python — обрабатывать исключение там, где оно было впервые перехвачено, не передавая его дальше). Задача решена — обработчик исключений в блоке “\_\_name\_\_ == '\_\_main\_\_'” теперь «знает», что в функции что-то случилось, и завершает работу с отличным от нуля кодом. Это может быть полезно при организации конвейеров из скриптов.

\*Тут мы подходим ко **второй причине** использования ключевого слова raise: поднять «[своё](https://docs.python.org/3.8/tutorial/errors.html#user-defined-exceptions)» (кастомное) исключение. Это уже продвинутый уровень программирования на Python. Покажем ознакомительный пример:

| class CalcError(Exception):  pass   def do\_calc(f\_path):  f = open(f\_path, 'a', encoding='utf-8')  try:  x = float(input('enter x val: '))  y = float(input('enter y val: '))  result = x / y  except ValueError as e:  print(f'wrong val: {e}')  raise CalcError  except ZeroDivisionError:  print('fault: Zero division')  raise CalcError  else:  f.write(f'{x} / {y} = {result}\n')  finally:  f.close()   if \_\_name\_\_ == '\_\_main\_\_':  f\_path = 'calc\_log.txt'  try:  do\_calc(f\_path)  except CalcError:  print(f'calc fail')  exit(1)  except Exception as e:  print(f'global error: {e}')  exit(2)  *# enter x val: 8 # enter y val: 0 # fault: Zero division # calc fail  # Process finished with exit code 1* |
| --- |

Что поменялось? Концепция обработки ошибок. Теперь если при вычислениях произошла какая-то ошибка, завершаем скрипт с кодом 1. Когда понадобится такой шаг? Когда кроме вызова функции do\_calc() будем делать что-то ещё (в этом примере этого нет), например, запрашивать имя пользователя перед вычислениями или имя файла для сохранения результатов. В дальнейшем при отладке проекта будем видеть: «что-то случилось с вычислениями», или «что-то не так при вводе имени пользователя», или «было введено неверное имя файла». Разумеется, придётся для этого создать соответствующие классы исключений по аналогии с классом CalcError — они должны быть унаследованы от Exception. Эти исключения вы будете поднимать в соответствующих функциях при помощи ключевого слова raise.

**\*Замечание:** после ключевого слова raise можно указывать объект исключения CalcError(), можно передать ему объект перехваченной ошибки CalcError(e), а можно просто указать класс выбрасываемого исключения CalcError (мы именно так и сделали в примере) — Python сам создаст объект. Рекомендуем внимательно изучить [пример](https://docs.python.org/3.8/tutorial/errors.html#raising-exceptions).

**Третья причина** использования ключевого слова raise — абстрагирование поведения кода от деталей. Особенно часто так делают в фреймворках, например Django. Конкретный пример. Есть HTML-форма, на которой пользователь вводит данные. Мы при обработке этой формы обнаруживаем ошибку в данных. Как предотвратить дальнейшие действия и сказать пользователю об ошибке? Если бы это был наш код, было бы проще: сделали бы специальный аргумент в некоторой функции или что-то подобное. Но это фреймворк, всё спрятано в «чёрном ящике». Как быть? Поднять специальное исключение [ValidationError](https://docs.djangoproject.com/en/3.1/ref/forms/validation/#raising-validation-error) с некоторыми аргументами:

| ...  ...  raise ValidationError('пароль слишком прост')  ... ... |
| --- |

И всё. Дальше фреймворк разберётся сам. Получается в некотором смысле [декларативное программирование](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5): мы не погружаемся в детали, а просто заявляем: «тут что-то не так». Можно воспринимать это как своего рода передачу сигнала на расстояние, в какую-то другую, возможно далёкую, часть кода.

**Четвёртая причина** использования ключевого слова raise — искусственно прервать выполнение кода, когда нет ошибки выполнения, но что-то идёт не так или, наоборот, нужные условия достигнуты. Например, нам нужно обработать несколько списков с числами и получить 5 нечётных чисел в случайном порядке:

| import random  class JobDone(Exception):  pass   def nums\_get(length, \*args):  nums = []  try:  for series in args:  while series:  random.shuffle(series)  num = series.pop()  if num % 2:  nums.append(num)  if len(nums) == length:  raise JobDone  except JobDone:  return nums   nums\_1 = [3, 6, 8, 9, 17] nums\_2 = [16, 22, 25] nums\_3 = [7, 11, 18] print(nums\_get(5, nums\_1, nums\_2, nums\_3)) *# [17, 3, 9, 25, 11]* |
| --- |

Можно было бы решить эту задачу через флаг — устанавливать его в правду во внутреннем цикле и проверять в конце внешнего:

| import random   def nums\_get(length, \*args):  nums = []  job\_done = False   for series in args:  while series:  random.shuffle(series)  num = series.pop()  if num % 2:  nums.append(num)  if len(nums) == length:  job\_done = True  break  if job\_done:  break  return nums   nums\_1 = [3, 6, 8, 9, 17] nums\_2 = [16, 22, 25] nums\_3 = [7, 11, 18] print(nums\_get(5, nums\_1, nums\_2, nums\_3)) *# [3, 9, 17, 25, 11]* |
| --- |

Чувствуете, насколько тяжелее читать такой код? А если будет ещё один внутренний цикл? Это ещё плюс проверки — код станет ещё запутаннее.

**Замечание**: не следует сильно переусердствовать с ключевым словом raise — обработка исключений происходит намного медленнее, чем проверка условия в конструкции if.

# 

# 

# Практическое задание

1. Написать скрипт, создающий стартер (заготовку) для проекта со следующей структурой **папок**:

| |--my\_project  |--settings  |--mainapp  |--adminapp  |--authapp |
| --- |

**Примечание:** подумайте о ситуации, когда некоторые папки уже есть на диске (как быть?); как лучше хранить конфигурацию этого стартера, чтобы в будущем можно было менять имена папок под конкретный проект; можно ли будет при этом расширять конфигурацию и хранить данные о вложенных папках и файлах (добавлять детали)?

1. \*(вместо 1) Написать скрипт, создающий из config.yaml стартер для проекта со следующей структурой:

| |--my\_project  |--settings  | |--\_\_init\_\_.py  | |--dev.py  | |--prod.py  |--mainapp  | |--\_\_init\_\_.py  | |--models.py  | |--views.py  | |--templates  | |--mainapp  | |--base.html  | |--index.html  |--authapp  | |--\_\_init\_\_.py  | |--models.py  | |--views.py  | |--templates  | |--authapp  | |--base.html  | |--index.html |
| --- |

**Примечание:** структуру файла config.yaml придумайте сами, его можно создать в любом текстовом редакторе «руками» (не программно); предусмотреть возможные исключительные ситуации, библиотеки использовать нельзя.

1. Создать структуру файлов и папок, как написано в задании 2 (при помощи скрипта или «руками» в проводнике). Написать скрипт, который собирает все шаблоны в одну папку templates, например:

| |--my\_project  ...  |--templates  | |--mainapp  | | |--base.html  | | |--index.html  | |--authapp  | |--base.html  | |--index.html |
| --- |

**Примечание:** исходные файлы необходимо оставить; обратите внимание, что html-файлы расположены в родительских папках (они играют роль пространств имён); предусмотреть возможные исключительные ситуации; это реальная задача, которая решена, например, во фреймворке django.

1. Написать скрипт, который выводит статистику для заданной папки в виде словаря, в котором ключи — верхняя граница размера файла (пусть будет кратна 10), а значения — общее количество файлов (в том числе и в подпапках), размер которых не превышает этой границы, но больше предыдущей (начинаем с 0), например:

| {  100: 15,  1000: 3,  10000: 7,  100000: 2  } |
| --- |

Тут 15 файлов размером не более 100 байт; 3 файла больше 100 и не больше 1000 байт...

Подсказка: размер файла можно получить из атрибута .st\_size объекта os.stat.

1. \*(вместо 4) Написать скрипт, который выводит статистику для заданной папки в виде словаря, в котором ключи те же, а значения — кортежи вида (<files\_quantity>, [<files\_extensions\_list>]), например:

| {  100: (15, ['txt']),  1000: (3, ['py', 'txt']),  10000: (7, ['html', 'css']),  100000: (2, ['png', 'jpg'])  } |
| --- |

Сохраните результаты в файл <folder\_name>\_summary.json в той же папке, где запустили скрипт.

Задачи со \* предназначены для продвинутых учеников, которым мало сделать обычное задание.

# Дополнительные материалы

1. [Лутц Марк. Изучаем Python](https://www.ozon.ru/context/detail/id/5730448/).
2. [Модуль shutil в Python](https://docs.python.org/3.8/library/shutil.html?highlight=shutil#module-shutil).
3. [Python Exceptions](https://realpython.com/python-exceptions/).

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://docs.python.org/3/>.