

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](#)

[Поиск файлов в папке](#)

[Полезные функции модуля os.path](#)

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

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

[Функция copyfileobj\(\)](#)

[Функция copyfile\(\)](#)

[Функции copy\(\) и copy2\(\)](#)

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

[Обработка исключительных ситуаций в Python](#)

[Блок try...except](#)

[Встроенные классы-исключения](#)

[Блок else](#)

[Блок finally](#)

[Часто используемые исключения](#)

[Ключевое слово raise](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

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\aiohhttp'
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](#) 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\(\)](#) 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\(\)](#) из модуля [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 слешы 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`: `isfile()`, `isdir()` или из `os`: `stat()`.

Например, покажем все папки в корне фреймворка [Django](#):

```
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](#) 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\(\)](#), которая возвращает генератор объектов [DirEntry](#). Это позволяет избежать дополнительных [системных вызовов](#) 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 0.07398164999999999
# 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 0.044809542999999995
# 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\(\)](#) (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\(\)`](#) — возвращает абсолютный путь;
- [`basename\(\)`](#) — возвращает имя файла из абсолютного или относительного пути;
- [`dirname\(\)`](#) — возвращает имя (путь) папки, в которой расположен файл;
- [`split\(\)`](#) — делит путь к файлу на путь к папке и имя файла (заменяет вызов двух предыдущих функций);
- [`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\(\)`](#) — склеивает путь из частей (надеюсь, вы не делаете это через строки!);
- [`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.listdir(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\(\)](#) 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\(\)](#):

```
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\(\)`](#) 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\(\)`](#):

```
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\(\)](#):

```
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\(\)](#) из модуля [shutil](#):

```
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\(\)](#) — копирование одного файлового объекта в другой;
- [copyfile\(\)](#) — копирование содержимого одного файла в другой (настройки доступа не копируются);
- [copy\(\)](#) — копирование файла (копируются настройки доступа);
- [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 простой 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](#), 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](#) 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](#) (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\(\)](#), а в `copy2()` — функция [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](#)). Начиная с Python 3.8, интерпретатор сам пытается использовать [наиболее эффективные](#) 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\(\)](#) 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](#) атрибута `__path__`. Теперь код будет работать «из коробки» — не надо вручную прописывать путь. Можете пока воспринимать этот трюк как магию.

Третье — использовали функцию [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](#). То есть мы не просим разрешения и не делаем проверок — пробуем выполнить код и имеем дело с последствиями. Для этого используется конструкция [try...except](#):

```
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](#). 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](#)», 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](#).

Очень важно почувствовать, какие возможности даёт вам такой подход к работе с исключениями. Как зум-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__` 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](#) (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](#). 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](#) — 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](#) — попытка обратиться к несуществующему атрибуту или методу;
- [IndexError](#) — ошибка индекса (в списке), обращение к элементу с индексом за пределами существующих;
- [KeyError](#) — ошибка ключа (в словаре), обращение к элементу словаря, которого в нём нет;
- [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 «[свое](#)» (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](#).

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](#) с некоторыми аргументами:

```
...
...
raise ValidationError('пароль слишком прост')
...
...
```

И всё. Дальше фреймворк разберётся сам. Получается в некотором смысле [декларативное программирование](#): 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)?

2. *(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.

3. 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`.

4. 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`.

5. *(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.](#)
2. [Модуль `shutil` в Python.](#)
3. [Python Exceptions.](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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