[4 Работа с файлами 1](#_Toc154433769)

[Открытие и закрытие файлов 1](#_Toc154433770)

[Текстовые файлы 2](#_Toc154433771)

[Файлы CSV 5](#_Toc154433772)

[Бинарные файлы 7](#_Toc154433773)

[Модуль shelve 8](#_Toc154433774)

[Модуль OS и работа с файловой системой 11](#_Toc154433775)

[Программа подсчета слов 12](#_Toc154433776)

[Запись и чтение архивных zip-файлов 13](#_Toc154433777)

[5 Работа с датами и временем 18](#_Toc154433778)

[Модуль datetime 18](#_Toc154433779)

[Операции с датами 20](#_Toc154433780)

**4 Работа с файлами**

**Открытие и закрытие файлов**

Python поддерживает множество различных типов файлов, но условно их можно разделить на два виде: текстовые и бинарные. Текстовые файлы - это к примеру файлы с расширением cvs, txt, html, в общем любые файлы, которые сохраняют информацию в текстовом виде. Бинарные файлы - это изображения, аудио и видеофайлы и т.д. В зависимости от типа файла работа с ним может немного отличаться.

При работе с файлами необходимо соблюдать некоторую последовательность операций:

1. Открытие файла с помощью метода **open()**
2. Чтение файла с помощью метода **read()** или запись в файл посредством метода **write()**
3. Закрытие файла методом **close()**

### Открытие и закрытие файла

Чтобы начать работу с файлом, его надо открыть с помощью функции **open()**, которая имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1 | open(file, mode) |

Первый параметр функции представляет путь к файлу. Путь файла может быть абсолютным, то есть начинаться с буквы диска, например, *C://somedir/somefile.txt*. Либо можно быть относительным, например, *somedir/somefile.txt* - в этом случае поиск файла будет идти относительно расположения запущенного скрипта Python.

Второй передаваемый аргумент - mode устанавливает режим открытия файла в зависимости от того, что мы собираемся с ним делать. Существует 4 общих режима:

* **r** (Read). Файл открывается для чтения. Если файл не найден, то генерируется исключение FileNotFoundError
* **w** (Write). Файл открывается для записи. Если файл отсутствует, то он создается. Если подобный файл уже есть, то он создается заново, и соответственно старые данные в нем стираются.
* **a** (Append). Файл открывается для дозаписи. Если файл отсутствует, то он создается. Если подобный файл уже есть, то данные записываются в его конец.
* **b** (Binary). Используется для работы с бинарными файлами. Применяется вместе с другими режимами - w или r.

После завершения работы с файлом его обязательно нужно закрыть методом close(). Данный метод освободит все связанные с файлом используемые ресурсы.

Например, откроем для записи текстовый файл "hello.txt":

|  |  |
| --- | --- |
| 1  2  3 | myfile = open("hello.txt", "w")    myfile.close() |

При открытии файла или в процессе работы с ним мы можем столкнуться с различными исключениями, например, к нему нет доступа и т.д. В этом случае программа выпадет в ошибку, а ее выполнение не дойдет до вызова метода close, и соответственно файл не будет закрыт.

В этом случае мы можем обрабатывать исключения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | try:      somefile = open("hello.txt", "w")      try:          somefile.write("hello world")      except Exception as e:          print(e)      finally:          somefile.close()  except Exception as ex:      print(ex) |

В данном случае вся работа с файлом идет во вложенном блоке try. И если вдруг возникнет какое-либо исключение, то в любом случае в блоке finally файл будет закрыт.

Однако есть и более удобная конструкция - конструкция **with**:

|  |  |
| --- | --- |
| 1  2 | with open(file, mode) as file\_obj:      инструкции |

Эта конструкция определяет для открытого файла переменную file\_obj и выполняет набор инструкций. После их выполнения файл автоматически закрывается. Даже если при выполнении инструкций в блоке with возникнут какие-либо исключения, то файл все равно закрывается.

Так, перепишем предыдущий пример:

|  |  |
| --- | --- |
| 1  2 | with open("hello.txt", "w") as somefile:      somefile.write("hello world") |

## Текстовые файлы

### Запись в текстовый файл

Чтобы открыть текстовый файл на запись, необходимо применить режим w (перезапись) или a (дозапись). Затем для записи применяется метод **write(str)**, в который передается записываемая строка. Стоит отметить, что записывается именно строка, поэтому, если нужно записать числа, данные других типов, то их предварительно нужно конвертировать в строку.

Запишем некоторую информацию в файл "hello.txt":

|  |  |
| --- | --- |
| 1  2 | with open("hello.txt", "w") as file:      file.write("hello world") |

Если мы откроем папку, в которой находится текущий скрипт Python, то увидем там файл hello.txt. Этот файл можно открыть в любом текстовом редакторе и при желании изменить.

Теперь дозапишем в этот файл еще одну строку:

|  |  |
| --- | --- |
| 1  2 | with open("hello.txt", "a") as file:      file.write("\ngood bye, world") |

Дозапись выглядит как добавление строку к последнему символу в файле, поэтому, если необходимо сделать запись с новой строки, то можно использовать эскейп-последовательность "\n". В итоге файл hello.txt будет иметь следующее содержимое:

|  |  |
| --- | --- |
| 1  2 | hello world  good bye, world |

Еще один способ записи в файл представляет стандартный метод **print()**, который применяется для вывода данных на консоль:

|  |  |
| --- | --- |
| 1  2 | with open("hello.txt", "a") as hello\_file:      print("Hello, world", file=hello\_file) |

Для вывода данных в файл в метод print в качестве второго параметра передается название файла через параметр file. А первый параметр представляет записываемую в файл строку.

### Чтение файла

Для чтения файла он открывается с режимом r (Read), и затем мы можем считать его содержимое различными методами:

* **readline()**: считывает одну строку из файла
* **read()**: считывает все содержимое файла в одну строку
* **readlines()**: считывает все строки файла в список

Например, считаем выше записанный файл построчно:

|  |  |
| --- | --- |
| 1  2  3 | with open("hello.txt", "r") as file:      for line in file:          print(line, end="") |

Несмотря на то, что мы явно не применяем метод readline() для чтения каждой строки, но в при переборе файла этот метод автоматически вызывается для получения каждой новой строки. Поэтому в цикле вручную нет смысла вызывать метод readline. И поскольку строки разделяются символом перевода строки "\n", то чтобы исключить излишнего переноса на другую строку в функцию print передается значение end="".

Теперь явным образом вызовем метод readline() для чтения отдельных строк:

|  |  |
| --- | --- |
| 1  2  3  4  5 | with open("hello.txt", "r") as file:      str1 = file.readline()      print(str1, end="")      str2 = file.readline()      print(str2) |

Консольный вывод:

hello world

good bye, world

Метод readline можно использовать для построчного считывания файла в цикле while:

|  |  |
| --- | --- |
| 1  2  3  4  5 | with open("hello.txt", "r") as file:      line = file.readline()      while line:          print(line, end="")          line = file.readline() |

Если файл небольшой, то его можно разом считать с помощью метода **read()**:

|  |  |
| --- | --- |
| 1  2  3 | with open("hello.txt", "r") as file:      content = file.read()      print(content) |

И также применим метод **readlines()** для считывания всего файла в список строк:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | with open("hello.txt", "r") as file:      contents = file.readlines()      str1 = contents[0]      str2 = contents[1]      print(str1, end="")      print(str2) |

При чтении файла мы можем столкнуться с тем, что его кодировка не совпадает с ASCII. В этом случае мы явным образом можем указать кодировку с помощью параметра **encoding**:

|  |  |
| --- | --- |
| 1  2  3 | filename = "hello.txt"  with open(filename, encoding="utf8") as file:      text = file.read() |

Теперь напишем небольшой скрипт, в котором будет записывать введенный пользователем массив строк и считывать его обратно из файла на консоль:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | # имя файла  FILENAME = "messages.txt"  # определяем пустой список  messages = list()    for i in range(4):      message = input("Введите строку " + str(i+1) + ": ")      messages.append(message + "\n")    # запись списка в файл  with open(FILENAME, "a") as file:      for message in messages:          file.write(message)    # считываем сообщения из файла  print("Считанные сообщения")  with open(FILENAME, "r") as file:      for message in file:          print(message, end="") |

Пример работы программы:

Введите строку 1: hello

Введите строку 2: world peace

Введите строку 3: great job

Введите строку 4: Python

Считанные сообщения

hello

world peace

great job

Python

## Файлы CSV

Одним из распространенных файловых форматов, которые хранят в удобном виде информацию, является формат **csv**. Каждая строка в файле csv представляет отдельную запись или строку, которая состоит из отдельных столбцов, разделенных запятыми. Собственно поэтому формат и называется Comma Separated Values. Но хотя формат csv - это формат текстовых файлов, Python для упрощения работы с ним предоставляет специальный встроенный модуль **csv**.

Рассмотрим работу модуля на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | import csv    FILENAME = "users.csv"    users = [      ["Tom", 28],      ["Alice", 23],      ["Bob", 34]  ]    with open(FILENAME, "w", newline="") as file:      writer = csv.writer(file)      writer.writerows(users)      with open(FILENAME, "a", newline="") as file:      user = ["Sam", 31]      writer = csv.writer(file)      writer.writerow(user) |

В файл записывается двухмерный список - фактически таблица, где каждая строка представляет одного пользователя. А каждый пользователь содержит два поля - имя и возраст. То есть фактически таблица из трех строк и двух столбцов.

При открытии файла на запись в качестве третьего параметра указывается значение newline="" - пустая строка позволяет корректно считывать строки из файла вне зависимости от операционной системы.

Для записи нам надо получить объект **writer**, который возвращается функцией csv.writer(file). В эту функцию передается открытый файл. А собственно запись производится с помощью метода writer.writerows(users) Этот метод принимает набор строк. В нашем случае это двухмерный список.

Если необходимо добавить одну запись, которая представляет собой одномерный список, например, ["Sam", 31], то в этом случае можно вызвать метод **writer.writerow(user)**

В итоге после выполнения скрипта в той же папке окажется файл users.csv, который будет иметь следующее содержимое:

|  |  |
| --- | --- |
| 1  2  3  4 | Tom,28  Alice,23  Bob,34  Sam,31 |

Для чтения из файла нам наоборот нужно создать объект **reader**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import csv    FILENAME = "users.csv"    with open(FILENAME, "r", newline="") as file:      reader = csv.reader(file)      for row in reader:          print(row[0], " - ", row[1]) |

При получении объекта reader мы можем в цикле перебрать все его строки:

Tom - 28

Alice - 23

Bob - 34

Sam - 31

### Работа со словарями

В примере выше каждая запись или строка представляла собой отдельный список, например, ["Sam", 31]. Но кроме того, модуль csv имеет специальные дополнительные возможности для работы со словарями. В частности, функция **csv.DictWriter()** возвращает объект writer, который позволяет записывать в файл. А функция **csv.DictReader()** возвращает объект reader для чтения из файла. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import csv    FILENAME = "users.csv"    users = [      {"name": "Tom", "age": 28},      {"name": "Alice", "age": 23},      {"name": "Bob", "age": 34}  ]    with open(FILENAME, "w", newline="") as file:      columns = ["name", "age"]      writer = csv.DictWriter(file, fieldnames=columns)      writer.writeheader()        # запись нескольких строк      writer.writerows(users)        user = {"name" : "Sam", "age": 41}      # запись одной строки      writer.writerow(user)    with open(FILENAME, "r", newline="") as file:      reader = csv.DictReader(file)      for row in reader:          print(row["name"], "-", row["age"]) |

Запись строк также производится с помощью методов writerow() и writerows(). Но теперь каждая строка представляет собой отдельный словарь, и кроме того, производится запись и заголовков столбцов с помощью метода **writeheader()**, а в метод csv.DictWriter в качестве второго параметра передается набор столбцов.

При чтении строк, используя названия столбцов, мы можем обратиться к отдельным значениям внутри строки: row["name"].

## Бинарные файлы

Бинарные файлы в отличие от текстовых хранят информацию в виде набора байт. Для работы с ними в Python необходим встроенный модуль **pickle**. Этот модуль предоставляет два метода:

* **dump(obj, file)**: записывает объект obj в бинарный файл file
* **load(file)**: считывает данные из бинарного файла в объект

При открытии бинарного файла на чтение или запись также надо учитывать, что нам нужно применять режим "b" в дополнение к режиму записи ("w") или чтения ("r"). Допустим, надо надо сохранить два объекта:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import pickle    FILENAME = "user.dat"    name = "Tom"  age = 19    with open(FILENAME, "wb") as file:      pickle.dump(name, file)      pickle.dump(age, file)    with open(FILENAME, "rb") as file:      name = pickle.load(file)      age = pickle.load(file)      print("Имя:", name, "\tВозраст:", age) |

С помощью функции dump последовательно записываются два объекта. Поэтому при чтении файла также последовательно посредством функции load мы можем считать эти объекты. Консольный вывод программы:

Имя: Tom Возраст: 28

Подобным образом мы можем сохранять и извлекать из файла наборы объектов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import pickle    FILENAME = "users.dat"    users = [      ["Tom", 28, True],      ["Alice", 23, False],      ["Bob", 34, False]  ]    with open(FILENAME, "wb") as file:      pickle.dump(users, file)      with open(FILENAME, "rb") as file:      users\_from\_file = pickle.load(file)      for user in users\_from\_file:          print("Имя:", user[0], "\tВозраст:", user[1], "\tЖенат(замужем):", user[2]) |

В зависимости от того, какой объект мы записывали функцией dump, тот же объект будет возвращен функцией load при считывании файла.

Консольный вывод:

Имя: Tom Возраст: 28 Женат(замужем): True

Имя: Alice Возраст: 23 Женат(замужем): False

Имя: Bob Возраст: 34 Женат(замужем): False

## Модуль shelve

Для работы с бинарными файлами в Python может применяться еще один модуль - **shelve**. Он сохраняет объекты в файл с определенным ключом. Затем по этому ключу может извлечь ранее сохраненный объект из файла. Процесс работы с данными через модуль shelve напоминает работу со словарями, которые также используют ключи для сохранения и извлечения объектов.

Для открытия файла модуль shelve использует функцию **open()**:

|  |  |
| --- | --- |
| 1 | open(путь\_к\_файлу[, flag="c"[, protocol=None[, writeback=False]]]) |

Где параметр flag может принимать значения:

* **c**: файл открывается для чтения и записи (значение по умолчанию). Если файл не существует, то он создается.
* **r**: файл открывается только для чтения.
* **w**: файл открывается для записи.
* **n**: файл открывается для записи Если файл не существует, то он создается. Если он существует, то он перезаписывается

Для закрытия подключения к файлу вызывается метод **close()**:

|  |  |
| --- | --- |
| 1  2  3 | import shelve  d = shelve.open(filename)  d.close() |

Либо можно открывать файл с помощью оператора **with**. Сохраним и считаем в файл несколько объектов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import shelve    FILENAME = "states2"  with shelve.open(FILENAME) as states:      states["London"] = "Great Britain"      states["Paris"] = "France"      states["Berlin"] = "Germany"      states["Madrid"] = "Spain"    with shelve.open(FILENAME) as states:      print(states["London"])      print(states["Madrid"]) |

Запись данных предполагает установку значения для определенного ключа:

|  |  |
| --- | --- |
| 1 | states["London"] = "Great Britain" |

А чтение из файла эквивалентно получению значения по ключу:

|  |  |
| --- | --- |
| 1 | print(states["London"]) |

В качестве ключей используются строковые значения.

При чтении данных, если запрашиваемый ключ отсутствует, то генерируется исключение. В этом случае перед получением мы можем проверять на наличие ключа с помощью оператора **in**:

|  |  |
| --- | --- |
| 1  2  3  4 | with shelve.open(FILENAME) as states:      key = "Brussels"      if key in states:          print(states[key]) |

Также мы можем использовать метод **get()**. Первый параметр метода - ключ, по которому следует получить значение, а второй - значение по умолчанию, которое возвращается, если ключ не найден.

|  |  |
| --- | --- |
| 1  2  3 | with shelve.open(FILENAME) as states:      state = states.get("Brussels", "Undefined")      print(state) |

Используя цикл for, можно перебрать все значения из файла:

|  |  |
| --- | --- |
| 1  2  3 | with shelve.open(FILENAME) as states:      for key in states:          print(key," - ", states[key]) |

Метод **keys()** возвращает все ключи из файла, а метод **values()** - все значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | with shelve.open(FILENAME) as states:        for city in states.keys():          print(city, end=" ")        # London Paris Berlin Madrid      print()      for country in states.values():          print(country, end=" ")     # Great Britain France Germany Spain |

Еще один метод **items()** возвращает набор кортежей. Каждый кортеж содержит ключ и значение.

|  |  |
| --- | --- |
| 1  2  3  4 | with shelve.open(FILENAME) as states:        for state in states.items():          print(state) |

Консольный вывод:

("London", "Great Britain")

("Paris", "France")

("Berlin", "Germany")

("Madrid", "Spain")

### Обновление данных

Для изменения данных достаточно присвоить по ключу новое значение, а для добавления данных - определить новый ключ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import shelve    FILENAME = "states2"  with shelve.open(FILENAME) as states:      states["London"] = "Great Britain"      states["Paris"] = "France"      states["Berlin"] = "Germany"      states["Madrid"] = "Spain"    with shelve.open(FILENAME) as states:        states["London"] = "United Kingdom"      states["Brussels"] = "Belgium"      for key in states:          print(key, " - ", states[key]) |

### Удаление данных

Для удаления с одновременным получением можно использовать функцию **pop()**, в которую передается ключ элемента и значение по умолчанию, если ключ не найден:

|  |  |
| --- | --- |
| 1  2  3  4 | with shelve.open(FILENAME) as states:        state = states.pop("London", "NotFound")      print(state) |

Также для удаления может применяться оператор **del**:

|  |  |
| --- | --- |
| 1  2  3 | with shelve.open(FILENAME) as states:        del states["Madrid"]    # удаляем объект с ключом Madrid |

Для удаления всех элементов можно использовать метод **clear()**:

|  |  |
| --- | --- |
| 1  2  3 | with shelve.open(FILENAME) as states:        states.clear() |

## Модуль OS и работа с файловой системой

Ряд возможностей по работе с каталогами и файлами предоставляет встроенный модуль **os**. Хотя он содержит много функций, рассмотрим только основные из них:

* **mkdir()**: создает новую папку
* **rmdir()**: удаляет папку
* **rename()**: переименовывает файл
* **remove()**: удаляет файл

### Создание и удаление папки

Для создания папки применяется функция **mkdir()**, в которую передается путь к создаваемой папке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import os    # путь относительно текущего скрипта  os.mkdir("hello")  # абсолютный путь  os.mkdir("[c://somedir](file:///C:\\somedir)")  os.mkdir("[c://somedir/hello](file:///C:\somedir\hello)") |

Для удаления папки используется функция **rmdir()**, в которую передается путь к удаляемой папке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import os    # путь относительно текущего скрипта  os.rmdir("hello")  # абсолютный путь  os.rmdir("[c://somedir/hello](file:///C:\somedir\hello)") |

### Переименование файла

Для переименования вызывается функция **rename(source, target)**, первый параметр которой - путь к исходному файлу, а второй - новое имя файла. В качестве путей могут использоваться как абсолютные, так и относительные. Например, пусть в папке *C://SomeDir/* располагается файл *somefile.txt*. Переименуем его в файл "hello.txt":

|  |  |
| --- | --- |
| 1  2  3 | import os    os.rename("[C://SomeDir/somefile.txt](file:///C:\SomeDir\somefile.txt)", "[C://SomeDir/hello.txt](file:///C:\SomeDir\hello.txt)") |

### Удаление файла

Для удаления вызывается функция **remove()**, в которую передается путь к файлу:

|  |  |
| --- | --- |
| 1  2  3 | import os    os.remove("[C://SomeDir/hello.txt](file:///C:\SomeDir\hello.txt)") |

### Существование файла

Если мы попытаемся открыть файл, который не существует, то Python выбросит исключение FileNotFoundError. Для отлова исключения мы можем использовать конструкцию try...except. Однако можно уже до открытия файла проверить, существует ли он или нет с помощью метода **os.path.exists(path)**. В этот метод передается путь, который необходимо проверить:

|  |  |
| --- | --- |
| 1  2  3  4  5 | filename = input("Введите путь к файлу: ")  if os.path.exists(filename):      print("Указанный файл существует")  else:      print("Файл не существует") |

## Программа подсчета слов

Рассмотрим работу со строками на небольшом примере, который будет представлять программу подсчета слов.

Пусть весь код программы будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | # Программа подсчета слов в файле  import os      def get\_words(filename):      with open(filename, encoding="utf8") as file:          text = file.read()      text = text.replace("\n", " ")      text = text.replace(",", "").replace(".", "").replace("?", "").replace("!", "")      text = text.lower()      words = text.split()      words.sort()      return words      def get\_words\_dict(words):      words\_dict = dict()        for word in words:          if word in words\_dict:              words\_dict[word] = words\_dict[word] + 1          else:              words\_dict[word] = 1      return words\_dict      def main():      filename = input("Введите путь к файлу: ")      if not os.path.exists(filename):          print("Указанный файл не существует")      else:          words = get\_words(filename)          words\_dict = get\_words\_dict(words)          print(f"Кол-во слов: {len(words)}")          print(f"Кол-во уникальных слов: {len(words\_dict)}")          print("Все использованные слова:")          for word in words\_dict:              print(word.ljust(20), words\_dict[word])      if \_\_name\_\_ == "\_\_main\_\_":      main() |

Здесь в функции get\_words() производится начальная сегментация текста на слова. Пи этом все пунктуационные знаки удаляются, а переводы стоки заменяется на пробелы. Затем происходит разбитие текста на слова. В качестве разделителя по умолчанию применяется пробел.

Далее в функции get\_words\_dict() получаем словарь из слов, где ключ - это уникальное слово, а значение - количество вхождений данного слова в тексте.

В функции main осуществляется ввод пути к файлу и вызов выше определенных функций, а также вывод все статистики.

Консольный вывод программы:

Введите путь к файлу: C:\SomeDir\hello.txt

Кол-во слов: 66

Кол-во уникальных слов: 54

Все использованные слова:

благодетель 2

в 1

всего 1

вы 1

горчичным 1

ее 1

ежели 3

еще 1

.......................

## Запись и чтение архивных zip-файлов

**Zip** представляет наиболее популярный формат архивации и сжатия файлов. И язык Python имеет встроенный модуль для работы с ними - **zipfile**. С помощью этого модуля можно создавать, считывать, записывать zip-файлы, получать их содержимое и добавлять в них файлы. Также поддерживается шифрование, но не поддерживается дешифрование.

Для представления zip-файла в этом модуле определен класс **ZipFile**. Он имеет следующий конструктор:

|  |  |
| --- | --- |
| 1 | ZipFile(file, mode='r', compression=ZIP\_STORED, allowZip64=True, compresslevel=None, \*, strict\_timestamps=True, metadata\_encoding=None) |

Параметры:

* file: путь к zip-файлу
* mode: режим открытия файла. Может принимать следующие значения:
  + r: применяется для чтения существующего файла
  + w: применяется для записи нового файла
  + a: применяется для добавления в файл
* compression: тип сжатия файла при записи. Может принимать значения:
  + ZIP\_STORED: архивация без сжатия (значение по умолчанию)
  + ZIP\_DEFLATED: стандартный тип сжатия при архивации в zip
  + ZIP\_BZIP2: сжатие с помощью способа BZIP2
  + ZIP\_LZMA: сжатие с помощью способа LZMA
* allowZip64: если равно True, то zip-файл может быть больше 4 Гб
* compresslevel: уровень сжатия при записи файла. Для типов сжатия ZIP\_STORED и ZIP\_LZMA не применяется. Для типа ZIP\_DEFLATED допустимые значения от 0 до 9, а для типа ZIP\_BZIP2 допустимые значения от 1 до 9.
* strict\_timestamps: при значении False позволяет работать с zip-файлами, созданными ранее 01.01.1980 и позже 31.12.2107
* metadata\_encoding: применяется для декодирования метаданных zip-файла (например, коментариев)

Для работы с файлами этот класс предоставляет ряд методов:

* **close()**: закрывает zip-файл
* **getinfo()**: возвращает информацию об одном файле из архива в виде объекта ZipInfo
* **namelist()**: возвращает список файлов архива
* **infolist()**: возвращает информацию обо всех файлах из архива в виде списока объектов ZipInfo
* **open()**: предоставляет доступ к одному из файлов в архиве
* **read()**: считывает файл из архива в набор байтов
* **extract()**: извлекает из архива один файл
* **extractall()**: извлекает все элементы из архива
* **setpassword()**: устанавливает пароль для zip-файла
* **printdir()**: выводит на консоль содержимое архива

### Создание и закрытие файла

Для создания архивного файла в конструктор ZipFile передается режим "w" или "a":

|  |  |
| --- | --- |
| 1  2  3 | from zipfile import ZipFile    myzip = ZipFile("metanit.zip", "w") |

После выполнения кода в текущей папке будет создаваться пустой архивный файл "metanit.zip".

После окончания работы с архивом для его закрытия применяется метод close():

|  |  |
| --- | --- |
| 1  2  3  4 | from zipfile import ZipFile    myzip = ZipFile("metanit.zip", "w")  myzip.close() |

Но так как ZipFile также представляет менеджер контекста, то он поддерживает выражение **with**, которое определяет контекст и автоматически закрывает файл по завершению контекста:

|  |  |
| --- | --- |
| 1  2  3  4 | from zipfile import ZipFile    with ZipFile("metanit.zip", "w") as myzip:      pass |

### Запись файлов в архив

Для записи файлов в архив применяется файл **write()**:

|  |  |
| --- | --- |
| 1 | write(filename, arcname=None, compress\_type=None, compresslevel=None) |

Первый параметр представляет файл, который записиывается в архив. Второй параметр - arcname устанавливает произвольное имя для файла внутри архива (по умолчанию это само имя файла). Третий параметр - compress\_type представляет тип сжатия, а параметр compresslevel - уровень сжатия.

Например, запишем в архив "metanit.zip" файл "hello.txt" (который, как предполагается, находится в той же папке, где и текущий скрипт python):

|  |  |
| --- | --- |
| 1  2  3  4 | from zipfile import ZipFile    with ZipFile("metanit.zip", "w") as myzip:      myzip.write("hello.txt") |

Стоит учитывать, что при открытии файла в режиме "w" при всех последующих записях текущее содержимое будет затираться, то есть фактически архивный файл будет создаваться заново. Если нам необходимо добавить, то необходимо определять zip-файл в режиме "a":

|  |  |
| --- | --- |
| 1  2  3  4  5 | from zipfile import ZipFile    with ZipFile("metanit.zip", "a") as myzip:      myzip.write("hello2.txt")      myzip.write("forest.jpg") |

Стоит отметить, что по умолчанию сжатие не применяется. Но при необходимости можно применить какой-нибудь способ сжатия и уровень сжатия"

|  |  |
| --- | --- |
| 1  2  3  4 | from zipfile import ZipFile, ZIP\_DEFLATED    with ZipFile("metanit.zip", "w", compression=ZIP\_DEFLATED, compresslevel=3) as myzip:      myzip.write("hello.txt") |

Необходимо учитывать, что если мы попробуем добавить в архив файлы с уже имеющимися именами, то консоль выведет предупреждение. Чтобы избежать наличия файлов с дублирующимися именами можно через второй папаметр метода write явным образом определить для них уникальное имя внутри архива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | from zipfile import ZipFile    with ZipFile("metanit.zip", "a") as myzip:      myzip.write("hello.txt", "hello1.txt")      myzip.write("hello.txt", "hello2.txt")      myzip.write("hello.txt", "hello3.txt") |

### Получение информации о файлах в архиве

Метод **infolist()** возвращает информацию о файлах в архиве с виде списка, где каждый отдельный файл представлен объектом ZipInfo:

|  |  |
| --- | --- |
| 1  2  3  4 | from zipfile import ZipFile    with ZipFile("metanit.zip", "a") as myzip:      print(myzip.infolist()) |

Класс **ZipInfo** предоставляет ряд атрибутов для хранения информации о файле. Основные из них:

* filename: название файла
* date\_time: дата и время последнего изменения файла в виде кортежа в формате (год, месяц, день, час, минута, секунда)
* compress\_type: тип сжатия
* compress\_size: размер после сжатия
* file\_size: оригинальный размер файла до сжатия

Получим эти данные по каждому отдельному файлу в архиве:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from zipfile import ZipFile    with ZipFile("metanit.zip", "r") as myzip:      for item in myzip.infolist():          print(f"File Name: {item.filename} Date: {item.date\_time} Size: {item.file\_size}") |

Примерный консольный вывод:

File Name: hello.txt Date: (2022, 11, 23, 20, 21, 34) Size: 18

File Name: forest.jpg Date: (2022, 11, 19, 20, 46, 52) Size: 103956

File Name: hello1.txt Date: (2022, 11, 23, 20, 21, 34) Size: 18

File Name: hello2.txt Date: (2022, 11, 23, 20, 21, 34) Size: 18

File Name: hello3.txt Date: (2022, 11, 23, 20, 21, 34) Size: 18

С помощью метода is\_dir() можно проверить, является ли элемент в архиве папкой:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | from zipfile import ZipFile    with ZipFile("metanit.zip", "r") as myzip:      for item in myzip.infolist():          if(item.is\_dir()):              print(f"Папка: {item.filename}")          else:              print(f"Файл: {item.filename}") |

Если надо получить только список имен входящих в архив файлов, то применяется метод **namelist()**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from zipfile import ZipFile    with ZipFile("metanit.zip", "r") as myzip:      for item in myzip.namelist():          print(item) |

Консольный вывод в моем случае:

hello.txt

forest.jpg

hello1.txt

hello2.txt

hello3.txt

С помощью метода **getinfo()** можно получить данные по одному из архивированных файлов, передав в метод его имя в архиве. Результат метода - объект ZipInfo:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | from zipfile import ZipFile    with ZipFile("metanit.zip", "r") as myzip:      try:          hello\_file = myzip.getinfo("hello.txt")          print(hello\_file.file\_size)      except KeyError:          print("Указанный файл отсутствует") |

Если в архиве не окажется элемента с указанным именем, то метод сгенерирует ошибку KeyError.

### Извлечение файлов из архива

Для извлечения всех файлов из архива применяется метод **extractall()**:

|  |  |
| --- | --- |
| 1 | extractall(path=None, members=None, pwd=None) |

Первый параметр метода устанавливает каталог для извлечения архива (по умолчанию извлечение идет в текущий каталог). Параметр members представляет список строк - список названий файлов, которые надо извлечт из архива. И третий параметр - pwd представляет пароль, в случае если архив закрыт паролем.

Например, извлечем все файлы из архива:

|  |  |
| --- | --- |
| 1  2  3  4 | from zipfile import ZipFile    with ZipFile("metanit.zip", "r") as myzip:      myzip.extractall() |

Извлечение в определенную папку:

|  |  |
| --- | --- |
| 1 | myzip.extractall(path="metanit") |

Извлечение части файлов:

|  |  |
| --- | --- |
| 1  2 | # извлекаем файлы  "hello.txt", "forest.jpg" в папку "metanit2"  myzip.extractall(path="metanit2", members=["hello.txt", "forest.jpg"]) |

Для извлечения одного файла применяется метод **extract()**, в который в качестве обязательного параметра передается имя извлекаемого файла:

|  |  |
| --- | --- |
| 1 | myzip.extract("hello.txt") |

### Считывание файла

Метод **read()** позволяет считать содержимое файла из архива в набор байтов:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from zipfile import ZipFile    with ZipFile("metanit.zip", "r") as myzip:      content = myzip.read("hello5.txt")      print(content) |

### Открытие файла

Метод **open()** позволяет открывать отдельные файлы из архива без непосредственного их извлечения:

|  |  |
| --- | --- |
| 1 | open(name, mode='r', pwd=None, \*, force\_zip64=False) |

В качестве первого обязательного параметра передается имя файла внутри архива. Второй параметр - mode устанавливает режим открытия. Параметр pwd задает пароль, если файл защищен паролем. И параметр force\_zip64 при значении True позволяет открывать файлы больше 4 Гб.

Этот файл может быть полезен для манипулирования файлом, например, для считывания его содержимого или, наоборот, для записи в него. Например, откроем файл и считаем его содержимое:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from zipfile import ZipFile    with ZipFile("metanit.zip", "a") as myzip:      # записываем в архив новый файл "hello5.txt"      with myzip.open("hello5.txt", "w") as hello\_file:          encoded\_str = bytes("Python...", "UTF- 8")          hello\_file.write(encoded\_str) |

# 5 Работа с датами и временем

## Модуль datetime

Основной функционал для работы с датами и временем сосредоточен в модуле **datetime** в виде следующих классов:

* **date**
* **time**
* **datetime**

### Класс date

Для работы с датами воспользуемся классом **date**, который определен в модуле datetime. Для создания объекта date мы можем использовать конструктор date, который последовательно принимает три параметра: год, месяц и день.

|  |  |
| --- | --- |
| 1 | date(year, month, day) |

Например, создадим какую-либо дату:

|  |  |
| --- | --- |
| 1  2  3  4 | import datetime    yesterday = datetime.date(2017,5, 2)  print(yesterday)      # 2017-05-02 |

Если необходимо получить текущую дату, то можно воспользоваться методом **today()**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from datetime import date    today = date.today()  print(today)      # 2017-05-03  print("{}.{}.{}".format(today.day, today.month, today.year))      # 2.5.2017 |

С помощью свойств day, month, year можно получить соответственно день, месяц и год

### Класс time

За работу с временем отвечает класс **time**. Используя его конструктор, можно создать объект времени:

|  |  |
| --- | --- |
| 1 | time([hour] [, min] [, sec] [, microsec]) |

Конструктор последовательно принимает часы, минуты, секунды и микросекунды. Все параметры необязательные, и если мы какой-то параметр не передадим, то соответствующее значение будет инициализироваться нулем.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | from datetime import time    current\_time = time()  print(current\_time)     # 00:00:00    current\_time = time(16, 25)  print(current\_time)     # 16:25:00    current\_time = time(16, 25, 45)  print(current\_time)     # 16:25:45 |

### Класс datetime

Класс **datetime** из одноименного модуля объединяет возможности работы с датой и временем. Для создания объекта datetime можно использовать следующий конструктор:

|  |  |
| --- | --- |
| 1 | datetime(year, month, day [, hour] [, min] [, sec] [, microsec]) |

Первые три параметра, представляющие год, месяц и день, являются обязательными. Остальные необязательные, и если мы не укажем для них значения, то по умолчанию они инициализируются нулем.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from datetime import datetime    deadline = datetime(2017, 5, 10)  print(deadline)     # 2017-05-10 00:00:00    deadline = datetime(2017, 5, 10, 4, 30)  print(deadline)     # 2017-05-10 04:30:00 |

Для получения текущих даты и времени можно вызвать метод **now()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | from datetime import datetime    now = datetime.now()  print(now)     # 2017-05-03 11:18:56.239443    print("{}.{}.{}  {}:{}".format(now.day, now.month, now.year, now.hour, now.minute))  # 3.5.2017  11:21    print(now.date())  print(now.time()) |

С помощью свойств day, month, year, hour, minute, second можно получить отдельные значения даты и времени. А через методы date() и time() можно получить отдельно дату и время соответственно.

### Преобразование из строки в дату

Из функциональности класса datetime следует отметить метод **strptime()**, который позволяет распарсить строку и преобразовать ее в дату. Этот метод принимает два параметра:

|  |  |
| --- | --- |
| 1 | strptime(str, format) |

Первый параметр str представляет строковое определение даты и времени, а второй параметр - формат, который определяет, как различные части даты и времени расположены в этой строке.

Для определения формата мы можем использовать следующие коды:

* **%d**: день месяца в виде числа
* **%m**: порядковый номер месяца
* **%y**: год в виде 2-х чисел
* **%Y**: год в виде 4-х чисел
* **%H**: час в 24-х часовом формате
* **%M**: минута
* **%S**: секунда

Применим различные форматы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | from datetime import datetime  deadline = datetime.strptime("22/05/2017", "%d/%m/%Y")  print(deadline)     # 2017-05-22 00:00:00    deadline = datetime.strptime("22/05/2017 12:30", "%d/%m/%Y %H:%M")  print(deadline)     # 2017-05-22 12:30:00    deadline = datetime.strptime("05-22-2017 12:30", "%m-%d-%Y %H:%M")  print(deadline)     # 2017-05-22 12:30:00 |

## Операции с датами

### Фоматирование дат и времени

Для форматирования объектов date и time в обоих этих классах предусмотрен метод **strftime(format)**. Этот метод принимает только один параметр, указывающий на формат, в который нужно преобразовать дату или время.

Для определения формата мы можем использовать один из следующих кодов форматирования:

* **%a**: аббревиатура дня недели. Например, Wed - от слова Wednesday (по умолчанию используются английские наименования)
* **%A**: день недели полностью, например, Wednesday
* **%b**: аббревиатура названия месяца. Например, Oct (сокращение от October)
* **%B**: название месяца полностью, например, October
* **%d**: день месяца, дополненный нулем, например, 01
* **%m**: номер месяца, дополненный нулем, например, 05
* **%y**: год в виде 2-х чисел
* **%Y**: год в виде 4-х чисел
* **%H**: час в 24-х часовом формате, например, 13
* **%I**: час в 12-ти часовом формате, например, 01
* **%M**: минута
* **%S**: секунда
* **%f**: микросекунда
* **%p**: указатель AM/PM
* **%c**: дата и время, отформатированные под текущую локаль
* **%x**: дата, отформатированная под текущую локаль
* **%X**: время, форматированное под текущую локаль

Используем различные форматы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from datetime import datetime  now = datetime.now()  print(now.strftime("%Y-%m-%d"))             # 2017-05-03  print(now.strftime("%d/%m/%Y"))             # 03/05/2017  print(now.strftime("%d/%m/%y"))             # 03/05/17  print(now.strftime("%d %B %Y (%A)"))        # 03 May 2017 (Wednesday)  print(now.strftime("%d/%m/%y %I:%M"))       # 03/05/17 01:36 |

При выводе названий месяцев и дней недели по умолчанию используются английские наименования. Если мы хотим использовать текущую локаль, то мы можем ее предварительно установить с помощью модуля locale:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | from datetime import datetime  import locale  locale.setlocale(locale.LC\_ALL, "")    now = datetime.now()  print(now.strftime("%d %B %Y (%A)"))        # 03 Май 2017 (среда) |

### Сложение и вычитание дат и времени

Нередко при работе с датами возникает необходимость добавить к какой-либо дате определенный промежуток времени или, наоборот, вычесть некоторый период. И специально для таких операций в модуле datetime определен класс **timedelta**. Фактически этот класс определяет некоторый период времени.

Для определения промежутка времени можно использовать конструктор timedelta:

|  |  |
| --- | --- |
| 1 | timedelta([days] [, seconds] [, microseconds] [, milliseconds] [, minutes] [, hours] [, weeks]) |

В конструктор мы последовательно передаем дни, секунды, микросекунды, миллисекунды, минуты, часы и недели.

Определим несколько периодов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | from datetime import timedelta    three\_hours = timedelta(hours=3)  print(three\_hours)       # 3:00:00  three\_hours\_thirty\_minutes = timedelta(hours=3, minutes=30)  # 3:30:00    two\_days = timedelta(2)  # 2 days, 0:00:00    two\_days\_three\_hours\_thirty\_minutes = timedelta(days=2, hours=3, minutes=30)  # 2 days, 3:30:00 |

Используя timedelta, мы можем складывать или вычитать даты. Например, получим дату, которая будет через два дня:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from datetime import timedelta, datetime    now = datetime.now()  print(now)                      # 2017-05-03 17:46:44.558754  two\_days = timedelta(2)  in\_two\_days = now + two\_days  print(in\_two\_days)              # 2017-05-05 17:46:44.558754 |

Или узнаем, сколько было времени 10 часов 15 минут назад, то есть фактически нам надо вычесть из текущего времени 10 часов и 15 минут:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from datetime import timedelta, datetime    now = datetime.now()  till\_ten\_hours\_fifteen\_minutes = now - timedelta(hours=10, minutes=15)  print(till\_ten\_hours\_fifteen\_minutes) |

### Свойства timedelta

Класс timedelta имеет несколько свойств, с помощью которых мы можем получить временной промежуток:

* **days**: возвращает количество дней
* **seconds**: возвращает количество секунд
* **microseconds**: возвращает количество микросекунд

Кроме того, метод **total\_seconds()** возвращает общее количество секунд, куда входят и дни, и собственно секунды, и микросекунды.

Например, узнаем какой временной период между двумя датами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | from datetime import timedelta, datetime    now = datetime.now()  twenty\_two\_may = datetime(2017, 5, 22)  period = twenty\_two\_may - now  print("{} дней  {} секунд   {} микросекунд".format(period.days, period.seconds, period.microseconds))  # 18 дней  17537 секунд   72765 микросекунд    print("Всего: {} секунд".format(period.total\_seconds()))  # Всего: 1572737.072765 секунд |

### Сравнение дат

Также как и строки и числа, даты можно сравнивать с помощью стандартных операторов сравнения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | from datetime import datetime    now = datetime.now()  deadline = datetime(2017, 5, 22)  if now > deadline:      print("Срок сдачи проекта прошел")  elif now.day == deadline.day and now.month == deadline.month and now.year == deadline.year:      print("Срок сдачи проекта сегодня")  else:      period = deadline - now      print("Осталось {} дней".format(period.days)) |