**Обработка ошибок и работа с датами**

Темы на сегодня:

* Виды ошибок
* Обработка ошибок
* Даты

ВИДЫ ОШИБОК

Ошибки – неотъемлемая часть написания программы. Важно понимать, почему они происходят – это, как правило, позволяет найти способ исправить их. Для удобства ошибки делятся на ряд типов:

* ZeroDivisionError – ошибка, возникающая при делении на ноль
* ImportError – ошибка импорта (не удалось подключить модуль или его атрибут)
* IndexError – индекс не входит в диапазон элементов
* KeyError – обращение к несуществующему ключу (в словаре, например)
* MemoryError – недостаточно памяти
* SyntaxError – синтаксическая ошибка (опечатка)
* TypeError – попытка применить операцию к объекту несоответствующего типа
* ValueError – получение аргумента правильного типа, но некорректного значения
* Warning – предупреждение

**Пример:**

*number1 = ‘1’*

*number2 = ‘1a’*

*float(number1) # 1*

*float(number1 # SyntaxError*

*int(number2) # TypeError*

*float(number2) # ValueError*

Ошибка описывается следующим образом:

*Тип ошибки*

*Где произошла ошибка (глобальная область)*

*Где произошла ошибка (локальная область 1) # Если имеется*

*...*

*Где произошла ошибка (локальная область n) # Если имеется*

*Расшифровка ошибки*

ОБРАБОТКА ОШИБОК

Для того, чтобы возможные ошибки были перехвачены, в Python существует специальная конструкция try-except. Структура следующая:

*try:*

*код, который мы хотим выполнить*

*except:*

*код, который нужно выполнить, если произошла ошибка*

**Пример:**

*number = ‘1a’*

*try:*

*print(float(number2))*

*except:*

*print(‘Ошибка’)*

*# Ошибка*

Можно отлавливать разные типы ошибок.

**Пример:**

*number = ‘1a’*

*try:*

*print(float(number2))*

*except TypeError:*

*print(‘Ошибка типа’)*

*except ValueError:*

*print(‘Ошибка значения’)*

*# Ошибка значения*

Однако, хотелось бы видеть ошибку без аварийного завершения работы программы. Для этого есть модуль **traceback**.

**Пример:**

*import traceback*

*number = ‘1a’*

*try:*

*print(float(number2))*

*except Exception:*

*print(traceback.print\_exc())*

*# Полное содержимое ошибки*

В определённых задачах также необходимо прописать обязательные действия, которые производятся после обработки ошибки. Для этого пишется дополнительный блок **finally**.

**Пример:**

*import traceback*

*number = ‘1a’*

*try:*

*print(float(number2))*

*except Exception:*

*print(traceback.print\_exc())*

*finally:*

*print(‘Завершили обработку ошибок’)*

*# Полное содержимое ошибки*

*# ‘Завершили обработку ошибок’*

ДАТЫ

Работа с датами – особый процесс в Python. По умолчанию для интерпретатора дата – такая же строка, как и любая. Поэтому существуют различные способы обработки дат. Самый популярный – с помощью библиотеки **datetime**. Импорт производится 2 способами:

* import datetime
* from datetime import datetime

Дело в том, что в модуле datetime есть библиотека datetime, которая и требуется. Если действовать первым способом, то нужно будет вызывать метод через datetime.datetime.метод(). При втором способе можно действовать проще: datetime.метод(). Оба варианта работают, но рекомендуется использовать второй способ, поскольку в таком случае легче ориентироваться и нет риска забыть продублировать название datetime.

Некоторые методы (полный список доступен в описании datetime):

* datetime.strptime(строка\_с\_датой, строка\_с\_форматом) – перевод даты из строки в объект datetime. Строка с форматом имеет следующие составляющие:
  + %a – день недели в сокращённом виде: Пн
  + %A – день недели в полном виде: Понедельник
  + %w – день недели в числовом виде. С воскресенья (0) по субботу (6)
  + %d – день месяца
  + %b – месяц в сокращённом виде: Янв
  + %B – месяц в полном виде: Январь
  + %m – месяц в числовом виде: 01
  + %y – год в сокращённом виде (без века): 19
  + %Y – год в полном виде: 2019
  + %H – часы в 24-часовом формате: 19
  + %I – часы в 12-часовом формате: 07 PM
  + %p – местный вариант наименования AM/PM
  + %M – минуты
  + %S – секунды
  + %f – микросекунды
  + %z – отклонение от Гринвича: +0300
  + %Z – название часового пояса

.strftime(строка\_с\_форматом) – обратная операция: перевод в строчный вид. Вообще, можно указать и конкретные значения. Например, 01 вместо %d. В таком случае в строке вместо дня будет это значение.

* .year – вернуть год
* .hour – вернуть час
* .weekday – вернуть день недели
* datetime.now() – вернуть дату + время в формате datetime

**Пример:**

*from datetime import datetime*

*line = ’19:00 18.05.19’*

*date = datetime.strptime(line, ‘%H:%M %d.%m.%Y’)*

*date.strftime(‘%H:%M %d.%m.%Y’) #’19:00 18.05.19’*

*date.year # 2019*

*datetime.now # datetime.datetime(2019, 12, 12, 12, 34, 53, 153869)*

Для изменения объектов datetime используется функция **timedelta**. Она задаёт нужный интервал, который нужно прибавить или отнять.

**Пример:**

*from datetime import datetime*

*from datetime import timedelta*

*line = ’19:00 18.05.19’*

*date = datetime.strptime(line, ‘%H:%M %d.%m.%Y’) #19:00 18.05.19*

*new\_date = date + timedelta(days=-5, minutes=8) #19:08 13.05.19*

На практике часто также используется и UNIX time – время (в секундах), которое прошло с 1 января 1970 года 00:00:00 UTC. Представляет собой целое число. Например, 1552251600. С такой формой представления удобно производить вычисления + занимает мало памяти. За работу с такой датой отвечает модуль **time**. На примере показано, как переводить в данный формат и обратно.

**Пример:**

*import time*

*from datetime import datetime*

*from datetime import date*

*date = date(2019, 3, 11)*

*unixtime = time.mktime(date.timetuple()) # Перевод в UNIX time: 1552251600*

*time = datetime.fromtimestamp(1552251600) # Обратный перевод*

*print(time) # datetime(2019, 3, 11, 0, 0)*

**Важно:** UNIX time не имеет часовых поясов, время всегда по Гринвичу, что необходимо учитывать при переводе.