Подробное введение в Python часть 4

1. Исключения в python. Конструкция try - except для обработки исключений

Исключения (exceptions) - ещё один тип данных в python. Исключения необходимы для того, чтобы сообщать программисту об ошибках.

Самый простейший пример исключения - деление на ноль:

```
>>> 100 / 0
Traceback (most recent call last):
  File "", line 1, in
    100 / 0
ZeroDivisionError: division by zero
```

Разберём это сообщение подробнее: интерпретатор нам сообщает о том, что он поймал исключение и напечатал информацию (Traceback (most recent call last)).

Далее имя файла (File ""). Имя пустое, потому что мы находимся в интерактивном режиме, строка в файле (line 1);

Выражение, в котором произошла ошибка (100 / 0).

Название исключения (ZeroDivisionError) и краткое описание исключения (division by zero).

Разумеется, возможны и другие исключения:

```
>>> 2 + '1'
Traceback (most recent call last):
  File "", line 1, in
    2 + '1'
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> int('qwerty')
Traceback (most recent call last):
  File "", line 1, in
    int('qwerty')
ValueError: invalid literal for int() with base 10: 'qwerty'
```

В этих двух примерах генерируются исключения TypeError и ValueError соответственно. Подсказки дают нам полную информацию о том, где порождено исключение, и с чем оно связано.

Рассмотрим иерархию встроенных в python исключений, хотя иногда вам могут встретиться и другие, так как программисты могут создавать собственные исключения. Данный список актуален для python 3.3, в более ранних версиях есть незначительные изменения.

- **BaseException** - базовое исключение, от которого берут начало все остальные.
  - **SystemExit** - исключение, порождаемое функцией sys.exit при выходе из программы.
  - **KeyboardInterrupt** - порождается при прерывании программы пользователем (обычно сочетанием клавиш Ctrl+C).
  - **GeneratorExit** - порождается при вызове метода close объекта generator.
  - **Exception** - а вот тут уже заканчиваются полностью системные исключения (которые лучше не трогать) и начинаются обыкновенные, с которыми можно работать.
    - **StopIteration** - порождается встроенной функцией next, если в итераторе больше нет элементов.
    - **ArithmeticError** - арифметическая ошибка.
      - **FloatingPointError** - порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.
      - **OverflowError** - возникает, когда результат арифметической операции слишком велик для представления. Не появляется при обычной работе с целыми числами (так как python поддерживает длинные числа), но может возникать в некоторых других случаях.
      - **ZeroDivisionError** - деление на ноль.
    - **AssertionError** - выражение в функции assert ложно.
    - **AttributeError** - объект не имеет данного атрибута (значения или метода).
    - **BufferError** - операция, связанная с буфером, не может быть выполнена.
    - **EOFError** - функция наткнулась на конец файла и не смогла прочитать то, что хотела.
    - **ImportError** - не удалось импортирование модуля или его атрибута.
    - **LookupError** - некорректный индекс или ключ.
      - **IndexError** - индекс не входит в диапазон элементов.
      - **KeyError** - несуществующий ключ (в словаре, множестве или другом объекте).
    - **MemoryError** - недостаточно памяти.
    - **NameError** - не найдено переменной с таким именем.
      - **UnboundLocalError** - сделана ссылка на локальную переменную в функции, но переменная не определена ранее.
    - **OSError** - ошибка, связанная с системой.
      - **BlockingIOError**
      - **ChildProcessError** - неудача при операции с дочерним процессом.
      - **ConnectionError** - базовый класс для исключений, связанных с подключениями.
        - **BrokenPipeError**
        - **ConnectionAbortedError**
        - **ConnectionRefusedError**
        - **ConnectionResetError**
      - **FileExistsError** - попытка создания файла или директории, которая уже существует.
      - **FileNotFoundError** - файл или директория не существует.
      - **InterruptedError** - системный вызов прерван входящим сигналом.
      - **IsADirectoryError** - ожидался файл, но это директория.
      - **NotADirectoryError** - ожидалась директория, но это файл.
      - **PermissionError** - не хватает прав доступа.
      - **ProcessLookupError** - указанного процесса не существует.
      - **TimeoutError** - закончилось время ожидания.
    - **ReferenceError** - попытка доступа к атрибуту со слабой ссылкой.

- **RuntimeError** - возникает, когда исключение не попадает ни под одну из других категорий.
- **NotImplementedError** - возникает, когда абстрактные методы класса требуют переопределения в дочерних классах.
- **SyntaxError** - синтаксическая ошибка.
  - **IndentationError** - неправильные отступы.
    - **TabError** - смешивание в отступах табуляции и пробелов.
- **SystemError** - внутренняя ошибка.
- **TypeError** - операция применена к объекту несоответствующего типа.
- **ValueError** - функция получает аргумент правильного типа, но некорректного значения.
- **UnicodeError** - ошибка, связанная с кодированием / раскодированием unicode в строках.
  - **UnicodeEncodeError** - исключение, связанное с кодированием unicode.
  - **UnicodeDecodeError** - исключение, связанное с декодированием unicode.
  - **UnicodeTranslateError** - исключение, связанное с переводом unicode.
- **Warning** - предупреждение.

Теперь, зная, когда и при каких обстоятельствах могут возникнуть исключения, мы можем их обрабатывать. Для обработки исключений используется конструкция try - except.

Первый пример применения этой конструкции:

```
>>>
```

```
>>> try:
...     k = 1 / 0
... except ZeroDivisionError:
...     k = 0
...
>>> print(k)
0
```

В блоке try мы выполняем инструкцию, которая может породить исключение, а в блоке except мы перехватываем их. При этом перехватываются как само исключение, так и его потомки. Например, перехватывая ArithmeticError, мы также перехватываем FloatingPointError, OverflowError и ZeroDivisionError.

```
>>> try:
...     k = 1 / 0
... except ArithmeticError:
...     k = 0
...
>>> print(k)
0
```

Также возможна инструкция except без аргументов, которая перехватывает вообще всё (и прерывание с клавиатуры, и системный выход и т. д.). Поэтому в такой форме инструкция except практически не используется, а используется except Exception. Однако чаще всего перехватывают исключения по одному,

для упрощения отладки (вдруг вы ещё другую ошибку сделаете, а except её перехватит).

Ещё две инструкции, относящиеся к нашей проблеме, это finally и else. Finally выполняет блок инструкций в любом случае, было ли исключение, или нет (применима, когда нужно непременно что-то сделать, к примеру, закрыть файл). Инструкция else выполняется в том случае, если исключения не было.

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

2. Файлы. Работа с файлами.

Прежде, чем работать с файлом, его надо открыть. С этим замечательно справится встроенная функция open:

```
f = open('text.txt', 'r')
```

У функции open много параметров, нам пока важны 3 аргумента: первый, это имя файла. Путь к файлу может быть относительным или абсолютным. Второй аргумент, это режим, в котором мы будем открывать файл.

Режим	Обозначение
'r'	открытие на чтение (является значением по умолчанию).
'w'	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый.

'x'	открытие на запись, если файла не существует, иначе исключение.
'a'	открытие на дозапись, информация добавляется в конец файла.
'b'	открытие в двоичном режиме.
't'	открытие в текстовом режиме (является значением по умолчанию).
'+'	открытие на чтение и запись

Режимы могут быть объединены, то есть, к примеру, 'rb' - чтение в двоичном режиме. По умолчанию режим равен 'rt'.

И последний аргумент, encoding, нужен только в текстовом режиме чтения файла. Этот аргумент задает кодировку.

Чтение из файла

Открыли мы файл, а теперь мы хотим прочитать из него информацию. Для этого есть несколько способов, но большого интереса заслуживают лишь два из них.

Первый - метод read, читающий весь файл целиком, если был вызван без аргументов, и n символов, если был вызван с аргументом (целым числом n).

```
>>> f = open('text.txt')
>>> f.read(1)
'H'
>>> f.read()
'ello world!\nThe end.\n\n'
```

Ещё один способ сделать это - прочитать файл построчно, воспользовавшись циклом for:

```
>>> f = open('text.txt')
>>> for line in f:
...     line
...
'Hello world!\n'
'\n'
'The end.\n'
'\n'
```

Запись в файл

Теперь рассмотрим запись в файл. Попробуем записать в файл вот такой вот список:

```
>>> l = [str(i)+str(i-1) for i in range(20)]
>>> l
['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98',
'109', '1110', '1211', '1312', '1413', '1514', '1615', '1716',
'1817', '1918']
```

Откроем файл на запись:

```
>>> f = open('text.txt', 'w')
```

Запись в файл осуществляется с помощью метода write:

```
>>> for index in l:
...     f.write(index + '\n')
...
4
3
3
3
3
```

Для тех, кто не понял, что это за цифры, поясню: метод write возвращает число записанных символов.

После окончания работы с файлом его **обязательно нужно закрыть** с помощью метода close:

```
>>> f.close()
```

Теперь попробуем воссоздать этот список из получившегося файла. Откроем файл на чтение (надеюсь, вы поняли, как это сделать?), и прочитаем строки.

```
>>> f = open('text.txt', 'r')
>>> l = [line.strip() for line in f]
>>> l
['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98',
'109', '1110', '1211', '1312', '1413', '1514', '1615', '1716',
'1817', '1918']
```

```
>>> f.close()
```

Мы получили тот же список, что и был. В более сложных случаях (словарях, вложенных кортежей и т. д.) алгоритм записи придумать сложнее. Но это и не нужно. В python уже давно придумали средства, такие как pickle или json, позволяющие сохранять в файле сложные структуры.

3. With ... as - менеджеры контекста

Конструкция with ... as используется для оборачивания выполнения блока инструкций менеджером контекста. Иногда это более удобная конструкция, чем try...except...finally.

Синтаксис конструкции with ... as:

```
"with" expression ["as" target] ("," expression ["as" target])* ":"
    suite
```

Теперь по порядку о том, что происходит при выполнении данного блока:

1. Выполняется выражение в конструкции with ... as.
2. Загружается специальный метод __exit__ для дальнейшего использования.
3. Выполняется метод __enter__. Если конструкция with включает в себя слово as, то возвращаемое методом __enter__ значение записывается в переменную.
4. Выполняется suite.
5. Вызывается метод __exit__, причём неважно, выполнилось ли suite или произошло исключение. В этот метод передаются параметры исключения, если оно произошло, или во всех аргументах значение None, если исключения не было.

Если в конструкции with - as было несколько выражений, то это эквивалентно нескольким вложенным конструкциям:

```
with A() as a, B() as b:
    suite
```

эквивалентно

```
with A() as a:
    with B() as b:
        suite
```

Для чего применяется конструкция with ... as? Для гарантии того, что критические функции выполнятся в любом случае. Самый распространённый пример использования этой конструкции - открытие файлов. Конструкция with ... as, как правило, является более удобной и гарантирует закрытие файла в любом случае. Например:

```
with open('newfile.txt', 'w', encoding='utf-8') as g:
    d = int(input())
    print('1 / {} = {}'.format(d, 1 / d), file=g)
```

Гарантирует, что файл будет закрыт вне зависимости от того, что введёт пользователь.

Более подробно про работу с файлами в ПР9