Урок 1

Концепции хранения информации

Особенности хранения символов в памяти компьютера. Недостатки кодировки ASCII. Введение в кодировку Unicode. Unicode в Python 3. Конвертация байтов и строк — понятие и примеры. Ошибки преобразования.

Введение

В курсе «Python 1» слушатели изучили основы программирования на Python и особенности синтаксиса языка, научились решать небольшие задачи на практике.

Данный курс — продолжение «Python 1»: мы углубим знакомство с ООП, научимся взаимодействовать с базами данных, создавать графический интерфейс пользователя и многопоточные приложения, тестировать код. Узнаем, как обеспечить информационную безопасность при разработке приложений. Слушателям стоит быть готовыми к тому, что к многим базовым темам возвращаться мы не будем.

В курсе «Python 2» другой характер будут носить и практические задания. Слушателям предлагается реализовать полноценную клиент-серверную систему обмена сообщениями на Python, с каждой новой темой добавляя новый функционал в приложение.

Чтобы обеспечить клиент-серверное взаимодействие, требуется в первую очередь изучить основы сетевого взаимодействия и его реализацию на Python. Но перед этим обратим внимание на особенности хранения данных в памяти компьютера, поскольку они обрабатываются в любой программе и формат представления результата может различаться.

Особенности хранения символов в памяти компьютера

Для хранения числовых и текстовых данных в памяти устройства используются последовательности кодов. При этом любому числу соответствует число двоичной системы счисления. Правила перевода чисел в двоичную систему просты. Но компьютер не только выполняет вычислительные операции, но еще и обрабатывает текстовую и мультимедийную информацию, так что стоит разобраться и с хранением других видов символов.

Для хранения букв также принято использовать код. Так как алфавит — это последовательность букв, каждую их них можно снабдить кодом символа — целым числом. Его записывают в память вычислительного устройства, а при отображении конвертируют обратно — в связанный с ним символ. Чтобы отделить числовое представление от символьного, надо хранить метаданные, то есть информацию о типе данных, сохраненных в области памяти вычислительного устройства.

Таким образом, набор букв алфавита соответствует таблице кодирования, и каждый его символ обладает уникальным кодом. Но существует множество алфавитов, и возникает вопрос — как кодировать все доступные на компьютере алфавиты.

Недостатки кодировки ASCII

В 60-х годах XX века силами государственного института стандартизации в США (ANSI) был подготовлен проект таблицы кодов символов, примененный впоследствии ко всем ОС. Этот проект назвали стандартной кодовой системой для операций с данными в США (American Standard Code for Information Interchange), сокращенно — ASCII.

С учетом особенностей стандарта ASCII для кодирования каждого символа назначается 1 байт (8 бит) памяти компьютера, то есть 8 ячеек памяти, способные сохранить 256 (2^8) любых значений. Первый блок кодов (128) — это главный раздел таблицы, он хранит базовую информацию независимо от алфавита: латинские буквы, цифры десятичной системы счисления, знаки препинания, служебные операторы (переводы строки, отступы). Второй блок кодов (125-255 позиции) — второстепенная частью таблицы, набор кодов символов национальных алфавитных систем.

Ввиду большого многообразия национальных алфавитных систем реализовано множество вариантов расширенных ASCII-таблиц. Причем одному алфавиту может соответствовать ряд кодовых таблиц. Например, в русском языке распространение получили таблицы Windows-1251, а также Koi8-r.

Отсутствие унифицированного стандарта вызывает трудности. Текст, подготовленный в одной системе кодирования, зачастую попадает к получателю, который пытается прочесть его в другой кодировке и видит непонятный набор символов. Еще один недостаток однобайтового подхода к кодированию — нехватка диапазона позиций (128–255) второго блока кодов, поскольку в некоторых алфавитах символов много. И наконец, если необходимо одновременно использовать в тексте конструкции на нескольких языках, автор оказывается в затруднительном положении — сразу две таблицы использовать нельзя.

Резюмируем недостатки однобайтовых кодировок:

1. Можно одновременно работать лишь с 256 символами, причем во втором блоке реализованы коды не для всех необходимых символов.

2. Шрифты привязаны к определенной кодировке.

3. Сложно конвертировать между кодировками, символы частично теряются при преобразовании (отсутствующие заменяются на схожие).

4. Сложно переносить файлы между вычислительными устройствами под управлением различных операционных систем: необходимо использовать дополнительную программу-конвертер.

5. Нельзя работать с иероглифическими системами письма, которые невозможно реализовать в однобайтовой кодировке.

Все эти проблемы были решены кодировкой Unicode.

Введение в кодировку Unicode

Принципы Unicode

В основу реализации кодировки заложен принцип четкого отделения символов от их отображения в памяти вычислительного устройства и на экране. Предлагается термин «юникод-символ», который фигурирует только в рамках теории и соглашения людей, закрепленного стандартом. Любому символу Unicode соответствует целое неотрицательное число — кодовая позиция.

Например, символ **U+0410** является кодом заглавной буквы «А» в кириллице. Она может быть отображена в памяти вычислительного устройства или на экране различными способами, но независимо от страны или других факторов данный код всегда будет соответствовать этому символу.

Есть подход с инкапсуляцией — отделением представления от реализации. Можно снабжать символ неограниченным количеством представлений, при этом у него будет определенное число реализаций. Данный подход успешно применяется в разработке программ.

Таким образом, текст можно представить в виде набора юникод-символов, а затем переслать в любую точку планеты. Если там поддерживается стандарт Unicode, получатели поймут смысл послания — воспримут его так же, как отправитель.

Примеры слов и соответствующие наборы юникод-символов:

```
"Компьютер", \u041a\u043e\u043c\u043f\u044c\u044e\u0442\u0435\u0440

"Программа", \u041f\u0440\u043e\u0433\u0440\u0430\u043c\u043c\u0430
"Интернет", \u0418\u043d\u0442\u0435\u0440\u043d\u0435\u0442
```

Хоть юникод-коды и именуются символами, они не всегда соответствуют классическому пониманию этого термина. Это могут быть технические символы, операторы, пунктуационные маркеры, языковые теги.

Кодовое пространство Unicode

Это диапазон-последовательность кодовых точек, доступных для привязки символов. Включает 1 114 112 кодовых точек в диапазоне 0-10FFFF. Есть раздел кодового пространства, зарезервированный под специальные нужды, который не будет участвовать в присвоении значений. Остальные кодовые позиции доступны. По стандарту Unicode версии 10.0 (июнь 2017 года) зарегистрировано 136 690 кодов, и каждый привязан к определенному символу.

Чтобы упростить работу с кодировкой, все кодовое пространство системы Unicode поделили на 17 плоскостей. На данный момент задействовано только шесть. Любой символ стандарта описывается в виде комбинации трех параметров: кода, состоящего из букв и шестнадцатеричных цифр, уникального имени символа и его представления.

Комбинации параметров символов в системе Unicode:

```
U+0061, "LATIN SMALL LETTER A" - a
U+00E4, "LATIN SMALL LETTER A WITH DIAERESIS" - ä
U+0056, "LATIN CAPITAL LETTER V" - V
U+0026, "AMPERSAND" - &
U+003B, "SEMICOLON" - ;
```

В стандарте Unicode также определены кодировки символов для хранения в памяти вычислительного устройства, то есть способы представления кода символа в байтах.

О кодировках

При передаче данных по сети надо их конвертировать в набор байтов. Поэтому при использовании юникод-стандарта делаем это с последовательностью юникод-символов.

При этом для кодирования всей области кодовых позиций применяется ряд кодировок — например, UTF-8 и UTF-16. Они обеспечивают конвертацию без потери информации. Возможно и параллельное существование однобайтных кодировок, позволяющих зашифровать индивидуальный, но ограниченный диапазон юникод-спектра — не более 256 кодовых позиций (кодовых точек). В таких кодировочных системах поддерживаются таблицы, где любому значению байта сопоставляется определенный юникод-символ (например, таблица **CP1251.TXT**). Несмотря на явные недостатки, однобайтные системы кодирования могут быть удобными, особенно если речь идет о работе со значительными объемами моноязыковых данных в текстовом отображении.

Наибольшее распространение из кодировок Unicode получила **UTF-8**. Она заняла лидирующие позиции в 2008 году — прежде всего за счет экономичности и открытой сопоставимости с семибитной кодировкой **ASCII**. Кодирование цифр, латинских букв, знаков препинания, служебных операторов в

UTF-8, как и в **ASCII**, осуществляется с помощью одного байта. Символы многих национальных алфавитных систем (кроме иероглифических) реализованы 2-3 байтами.

Стоит отметить, что кодировка **UTF-8** имеет переменную длину кода. При этом любому юникод-символу сопоставляется набор кодовых квантов с минимальной длиной, равной одному кванту. Квант кода в битовом выражении — это 8 бит. Для кодировок, относящихся к системе **UTF-16**, данный параметр равняется 16 битам, а к **UTF-32** — 32 битам.

Строковые данные в приложениях хранятся в 16-битных кодировках благодаря простоте их использования, а также в силу того, что символы, относящиеся к главным мировым письменным системам, шифруются в виде шестнадцатибитового кванта. Например, язык программирования Java, наряду с ОС Windows, при реализации внутреннего отображения строк использует **UTF-16**.

Но при работе в системе Unicode формат хранения строковых данных в рамках конкретного приложения не представляет особой важности. Если он обеспечивает корректное кодирование всех — более миллиона — кодовых точек, а на границе работы приложения (при считывании данных из файла или их сохранении в буфере обмена) не возникают потери информации — такой подход полностью работоспособен и эффективен.

Основные характеристики Unicode

1. В основу стандарта Unicode заложен принцип отделения символов от их отображения в памяти вычислительного устройства и на экране монитора.

2. Символ системы Unicode не всегда отождествляется с символом в привычном понимании — с буквой, цифрой, знаком пунктуации, иероглифом.

3. Кодовое пространство стандарта включает 1 114 112 кодовых точек, находящихся в пределах **0-10FFFF**.

4. Основная многоязыковая плоскость содержит символы Unicode в промежутке **U+0000-U+FFFF**, кодируемые в **UTF-16** с помощью двух байтов.

5. Каждая кодировка Unicode обеспечивает кодирование всего пространства кодовых точек с возможностью преобразования между такими кодировочными системами без потерь данных.

6. Однобайтовые кодировки используются для незначительной части юникод-символов, но полезны при обработке больших объемов моноязыковых данных.

7. Кодировочные системы **UTF-8** и **UTF-16** характеризуются переменной длиной кода. В **UTF-8** возможно шифрование любого символа посредством одного, двух, трех или четырех байтов. В UTF-16 — с помощью двух или четырех байтов.

8. Формат отображения текстовой информации применительно к отдельному приложению — произвольный, если корректно используется все пространство кодовых точек Unicode и нет потерь в трансграничной отправке данных.

Unicode в Python 3

Концепции представления информации

Рассмотрим использование стандарта Unicode и языка Python 3. Человек, работая с компьютером, воспринимает текст, а само вычислительное устройство — представление данных в байтах. В Python 3 реализованы две концепции:

- Текст — неизменяемый набор юникод-символов типа **str** (строка). Более корректная трактовка понятия «текст» — неизменяемый набор кодов (**code points**) Unicode;
- Данные — неизменяемый набор байтов, имеющий тип **bytes** (байты).

Строки

Строка — последовательность кодов Unicode, может быть записана различными способами. Примеры строк (файл **examples/01_unicode_in_python3/strings.py**):

```
In[1]: progr_1 = 'Программирование'

In[2]: print(progr_1)
Out[2]: Программирование

In[3]: print(type(progr_1))
Out[3]: <class 'str'>

In[4]: progr_2 = 'Programování'

In[5]: print(progr_2)
Out[5]: Programování
```

Символ Unicode можно записать не в традиционном (буквенном или цифровом представлении), а с помощью имени символа (файл **examples/01_unicode_in_python3/strings.py**):

```
In[6]: unic_s_1= "\N{LATIN SMALL LETTER C WITH DOT ABOVE}"

In[7]: print(unic_s_1)
Out[7]: ċ
```

Или с помощью особого формата (файл **examples/01_unicode_in_python3/strings.py**):

```
In[8]: unic_s_2 = "\u010B"

In[9]: print(unic_s_2)
Out[9]: ċ
```

Так же и строка может быть представлена как последовательность юникод-кодов (файл **examples/01_unicode_in_python3/strings.py**):

```
In[10]: progr_3 = 'Программа'

In[11]: progr_4 = '\u041f\u0440\u043e\u0433\u0440\u0430\u043c\u043c\u0430'
```

```
In[12]: print(progr_4)
Out[12]: Программа

In[13]: print(progr_3 == progr_4)
Out[13]: True

In[14]: print(len(progr_4))
Out[14]: 9
```

Получить значение числового представления для определенного юникод-символа можно с помощью функции **ord** (файл **examples/01_unicode_in_python3/strings.py**):

```
In[15]: print(ord('ã'))
Out[15]: 227
```

И наоборот — чтобы узнать, какой символ скрывается за определенным кодом, следует указать команду **chr** (файл **examples/01_unicode_in_python3/strings.py**):

```
In[16]: print(chr(227))
Out[16]: ã
```

Байты

Имеют аналогичное строкам обозначение, но маркируются дополнительным указателем «**b**» в начале набора (файл **examples/01_unicode_in_python3/bytes.py**):

```
In[17]: bytes_s_1 = b'\u041f\u0440\u043e\u0433\u0440\u0430\u043c\u043c\u0430'

In[18]: bytes_s_2 = b"\u041f\u0440\u043e\u0433\u0440\u0430\u043c\u043c\u0430"

In[19]: bytes_s_3 = b'''\u041f\u0440\u043e\u0433\u0440\u0430\u043c\u043c\u0430'''

In[20]: print(type(bytes_s_1))
Out[20]:  <class 'bytes'>
```

В языке Python байты, соответствующие символам ASCII (например, буквам латинского алфавита), имеют внешнее представление как сама последовательность символов, а не как связанные с ними байты. Отличие в том, что байтовый тип обязательно содержит маркировку «**b**» (файл **examples/01_unicode_in_python3/bytes.py**):

```
In[21]: bytes_s_4 = b'Program'

In[22]: print(bytes_s_4)
Out[22]: b'Program'

In[23]: print(len(bytes_s_4))
```

```
Out[23]: 7
```

Если указать в байтовом типе данных символ, не относящийся к ASCII, появится сообщение об ошибке (файл **examples/01_unicode_in_python3/bytes.py**):

```
In[24]: bytes_s_5 = b'Программа'

In[25]: print(bytes_s_5)
Out[25]:
    File "C:\Users\Администратор\Desktop\Курс Питон 2.1\01.
    Концепции хранения информации\examples\01_unicode_in_python3\
    bytes.py", line 15
        bytes_s_5 = b'Программа'
                    ^
    SyntaxError: bytes can only contain ASCII literal characters.
```

Конвертация байтов и строк

Особенности конвертации

Байтовое представление данных выглядит не самым понятным образом для программиста. Но избежать работы с байтами практически невозможно, особенно если речь идет о взаимодействии с файловой системой и сетью, когда результат может возвращаться в байтовом представлении.

Байты надо преобразовывать в строковый формат или наоборот. Для этого применяется метод, определяющий направление преобразования (кодирование или декодирование), и кодировка как аргумент метода (ключ шифрования).

Чтобы зашифровать строку в набор байтов, применяется метод **encode** (файл **examples/02_bytes_and_string_convertations/encode_decode.py**):

```
In[26]: enc_str = 'Кодировка'

In[27]: enc_str_bytes = enc_str.encode('utf-8')

In[28]: print(enc_str_bytes)
Out[28]:
b'\xd0\x9a\xd0\xbe\xd0\xb4\xd0\xb8\xd1\x80\xd0\xbe\xd0\xb2\xd0\xba\xd0\xb0'
```

За выполнение обратного процесса отвечает метод **decode** (файл **examples/02_bytes_and_string_convertations/encode_decode.py**):

```
In[29]: dec_str_bytes =
b'\xd0\x9a\xd0\xbe\xd0\xb4\xd0\xb8\xd1\x80\xd0\xbe\xd0\xb2\xd0\xba\xd0\xb0'

In[30]: dec_str = dec_str_bytes.decode('utf-8')
```

```
In[31]: print(dec_str)
Out[31]: Кодировка
```

Encode (и другие методы работы со строковыми данными) реализован также для класса **str** (файл **examples/02_bytes_and_string_convertations/encode_decode.py**):

```
In[32]: str_1 = 'Программа'

In[33]: str_1_enc = str.encode(str_1, encoding='utf-8')

In[34]: print(str_1_enc)
Out[34]:
b'\xd0\x9f\xd1\x80\xd0\xbe\xd0\xb3\xd1\x80\xd0\xb0\xd0\xbc\xd0\xbc\xd0\xb0'
```

А **decode** (и другие методы работы с байтами) предусмотрен у класса **bytes** (файл **examples/02_bytes_and_string_convertations/encode_decode.py**):

```
In[35]: bytes_1 =
b'\xd0\x9f\xd1\x80\xd0\xbe\xd0\xb3\xd1\x80\xd0\xb0\xd0\xbc\xd0\xbc\xd0\xb0'

In[36]: bytes_1_enc = bytes.decode(bytes_1, encoding='utf-8')

In[37]: print(bytes_1_enc)
Out[37]: Программа
```

Эти методы содержат указание кодировки как ключевого аргумента или позиционного (файл **examples/02_bytes_and_string_convertations/encode_decode.py**):

```
In[38]: bytes_1 =
b'\xd0\x9f\xd1\x80\xd0\xbe\xd0\xb3\xd1\x80\xd0\xb0\xd0\xbc\xd0\xbc\xd0\xb0'

In[39]: bytes_1_enc = bytes.decode(bytes_1, 'utf-8')

In[40]: print(bytes_1_enc)
Out[40]: Программа
```

Порядок работы с Unicode и байтами

Алгоритм оперирования данными в формате Unicode и байтовом формате раскрывается с помощью правила «юникод-сэндвич»:

- Байты, считываемые приложением из памяти вычислительного устройства, как можно раньше конвертировать в Unicode (строковый формат);
- В рамках приложения оперировать с данными только в строковом формате;
- Осуществлять конвертацию строк в байты непосредственно перед передачей данных.

Примеры конвертации байтов и строк

Модуль subprocess

Результатом выполнения данной команды является последовательность байт (файл **examples/03_practical_bytes_and_string_convertations/modules.py**):

```
In[41]: import subprocess

In[42]: args = ['ping', 'google.com']

In[43]: subproc_ping = subprocess.Popen(args, stdout=subprocess.PIPE)

In[44]: for line in subproc_ping.stdout:
            print(line)
Out[44]: b'\x8e\xa1\xac\xa5\xad \xaf\xa0\xaa\xa5\xe2\xa0\xac\xa8...'
```

Чтобы обработать полученный результат, его необходимо конвертировать в строковый формат. При этом в цикле надо выполнить декодирование строки в байтовом выражении и перевести ее в Unicode (файл **examples/03_practical_bytes_and_string_convertations/modules.py**):

```
In[45]: for line in subproc_ping.stdout:
            print(line.decode('utf-8'))
Out[45]: UnicodeDecodeError: 'utf-8' codec can't decode byte 0x8e in position 0:
invalid start byte
```

Но в этом случае декодер **UTF-8** может сгенерировать исключение, так как байтовое значение **0x8e** является некорректным в этой кодировке. Это происходит из-за того, что выводимое в результате работы модуля **subprocess** сообщение было закодировано не с помощью **UTF-8**. Во избежание ошибок и кодирование, и декодирование данных следует выполнять в одной кодировке — **UTF-8**.

Чтобы исправить ошибку в примере, необходимо определить исходную кодировку сообщения, раскодировать с ее помощью результат работы модуля **subprocess** и перевести его в **UTF-8**.

На вычислительных устройствах под управлением русифицированной ОС Windows при запуске консольных приложений чаще всего используется кириллическая кодировка **cp866**, в которой и

закодирован выводимый результат работы модуля **subprocess**. Для других приложений — это **windows-1251** (файл **examples/03_practical_bytes_and_string_convertations/modules.py**):

```
In[46]: for line in subproc_ping.stdout:
            line = line.decode('cp866').encode('utf-8')
            print(line.decode('utf-8'))

Out[46]: Обмен пакетами с google.com [74.125.232.224] с 32 байтами данных:
Ответ от 74.125.232.224: число байт=32 время=36мс TTL=56
Ответ от 74.125.232.224: число байт=32 время=36мс TTL=56
Ответ от 74.125.232.224: число байт=32 время=36мс TTL=56
Ответ от 74.125.232.224: число байт=32 время=36мс TTL=56
Статистика Ping для 74.125.232.224:

    Пакетов: отправлено = 4, получено = 4, потеряно = 0
    (0% потерь)
Приблизительное время приема-передачи в мс:
    Минимальное = 36 мсек, Максимальное = 36 мсек, Среднее = 36 мсек
```

В данном примере при переборе строк результата работы модуля **subprocess** выполняется конвертация каждой из строк в формат кодировки **cp866**, после чего результат переформатируется в **UTF-8**. При этом он представляет собой набор кодов Unicode (байтовый формат). Для дальнейшей работы с результатом как со строкой необходимо преобразовать его в этот тип, то есть выполнить операцию **decode**.

Последовательность работы с данными:

1. Байтовый формат **cp866** -> строка в формате **cp866**.
2. Строка в формате **cp866** -> байтовый формат **UTF-8**.
3. Байтовый формат **UTF-8** -> строка в формате **UTF-8**.

Модуль telnetlib

Модуль **telnetlib** предоставляет класс **Telnet**, реализующий протокол **Telnet** и позволяющий пользователю работать с удаленным компьютером, как со своим.

В различных модулях конвертация строк и байтов либо выполняется в автоматическом режиме, либо может потребоваться указать явное преобразование.

Особенность модуля **telnetlib** — необходимость передачи данных в байтах, а не в строках при работе с методами **read_until** и **write**. Поскольку возвращаемый результат также представляет собой байты, надо обязательно выполнить декодирование (файл **examples/03_practical_bytes_and_string_convertations/modules.py**):

```
import telnetlib
import time

tn_connect = telnetlib.Telnet('10.0.0.1')

tn_connect.read_until(b'Username:')
tn_connect.write(b'user\n')

t.read_until(b'Password:')
t.write(b'pass\n')

time.sleep(5)

output = tn_connect.read_very_eager().decode('cp866').encode('utf-8')
print(output.decode('utf-8'))
```

Работа с файловой системой

Чтобы обратиться к определенному файлу и прочесть его содержимое, применяется следующая конструкция:

```
with open(file_name) as f_n:
    for el_str in f_n
        print(el_str)
```

На практике при чтении из файла извлекаются данные, которые автоматически преобразуются в строковое представление. При этом используется кодировка по умолчанию. Для русскоязычных версий ОС Windows это, как правило, **cp1251** (файл **examples/03_practical_bytes_and_string_convertations/file_system.py**):

```
In[47]: import locale

In[48]: def_coding = locale.getpreferredencoding()

In[48]: print(def_coding)
Out[48]: cp1251
```

При работе с файлами также можно определить наименование кодировки, которая будет использоваться при операциях с ними (файл **examples/03_practical_bytes_and_string_convertations/file_system.py**):

```
In[49]: f_n = open("test.txt", "w")

In[50]: f_n.write("test test test")

In[51]: f_n.close()

In[52]: print(f_n)
Out[52]:  <_io.TextIOWrapper name='test.txt' mode='w' encoding='cp1251'>
```

Но при выполнении операций с файловой системой более корректная практика — явно указывать кодировку, поскольку она может различаться в ОС (файл **examples/03_practical_bytes_and_string_convertations/file_system.py**):

```
In[53]: with open('test.txt', encoding='utf-8') as f_n:

In[54]: for el_str in f_n:
            print(el_str, end='')
Out[54]:  test test test
```

Таким образом, в модулях по-разному реализована конвертация строк и байтов. Отдельные функции этих методов ожидают передачу аргументов и возвращают результат в виде одного типа данных (строкового или байтового). Для других формат данных на входе и выходе может различаться.

Ошибки преобразования

Варианты ошибок

При выполнении преобразований строковых данных и последовательностей байтов необходимо точно знать наименование кодировки и ее возможности, иначе возможны ошибки, связанные с конвертацией:

1. ***Отсутствие в кодировке механизма преобразования данных из одного формата в другой.***

   Например, в кодировке ASCII не предусмотрено преобразование кириллицы в байты (файл **examples/04_converation_errors/errors_variants.py**):

```
In[55]:  err_str_1 = 'Программа'

In[56]:  print(err_str.encode('ascii'))
Out[56]: UnicodeEncodeError: 'ascii' codec can't encode characters in position
0-8: ordinal not in range(128)
```

   Строку в байтах преобразовать в строковый формат с помощью кодировки ASCII тоже будет невозможно — программа выдаст ошибку (файл **examples/04_converation_errors/errors_variants.py**):

```
In[57]:  err_str_2 = 'Программа'

In[58]:  err_str_2_bytes = err_str_2.encode('utf-8')

In[59]:  err_str_2_str = err_str_2_bytes.decode('ascii')

In[60]:  print(err_str_2_str)
Out[60]: UnicodeDecodeError: 'ascii' codec can't decode byte 0xd0 in position 0:
ordinal not in range(128)
```

2. ***Использование при конвертации различных кодировок.***

   Речь о том случае, когда кодирование осуществляется в привязке к одной кодировке, а декодирование — к другой (файл **examples/04_converation_errors/errors_variants.py**):

```
In[61]: err_str_3 = 'Testování'

In[62]: utf_16_bytes = err_str_3.encode('utf-16')

In[63]: utf_8_str = utf_16_bytes.decode('utf-8')

In[64]: print(utf_8_str)
Out[64]: UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 0:
invalid start byte
```

Механизмы обработки ошибок

У методов **encode** и **decode** есть режимы обработки ошибок, которые указывают, как реагировать на ошибку преобразования.

Обработка ошибок метода encode

При использовании метода **encode** при возникновении ошибок генерируется исключение **UnicodeError**. Например в случае, рассмотренном выше, — где невозможно преобразовать кириллицу в байты с помощью кодировки ASCII.

Чтобы решить подобную проблему и запретить генерацию исключения **UnicodeError**, можно использовать режим **replace** для замены недостающих символов знаком вопроса (файл **examples/04_converation_errors/error_handling_mechanisms.py**):

```
In[65]: handl_err = 'Testování'

In[66]: handl_err_bytes = handl_err.encode('ascii', 'replace')

In[67]: print(handl_err_bytes)
Out[67]:  b'Testov?n?'
```

Можно применить метод **namereplace** для замены символа именем. Данная возможность доступна в Python, начиная с версии 3.5 (файл **examples/04_converation_errors/error_handling_mechanisms.py**):

```
In[68]: handl_err_bytes_2 = handl_err.encode('ascii', 'namereplace')

In[69]: print(handl_err_bytes_2)
Out[69]:  b'Testov\\N{Latin Small Letter a with Acute}n\\N{ Latin Small Letter i
with Acute }'
```

Еще вариант — просто проигнорировать символы, у которых есть проблемы с кодированием. Для этого применяется режим **ignore** (файл **examples/04_converation_errors/error_handling_mechanisms.py**):

```
In[70]: handl_unicode = 'Testování'

In[71]: handl_bytes = handl_unicode.encode('ascii', 'ignore')

In[72]: print(handl_bytes)
Out[72]:  b'Testovn'
```

Обработка ошибок метода decode

При некорректном декодировании генерируется исключение **UnicodeDecodeError**, которое может быть заблокировано механизмами **ignore** и **replace**. Они функционируют аналогично таким же в методе **encode** (файл **examples/04_converation_errors/error_handling_mechanisms.py**):

```
In[73]: handl_str = 'Testování'

In[74]: handl_str_utf8 = handl_str.encode('utf-8')

In[75]: print(handl_str_utf8)
Out[75]:  b'Testov\xc3\xa1n\xc3\xad'

In[76]: handl_str_utf8_str = handl_str_utf8.decode('ascii', 'ignore')

In[77]:  print(handl_str_utf8_str)
Out[77]:  Testovn
```

```
In[78]:  handl_str = 'Testování'

In[79]:  handl_str_utf8 = handl_str.encode('utf-8')

In[80]:  handl_str_utf8_str = handl_str_utf8.decode('ascii', 'replace')

In[81]:  print(handl_str_utf8_str)
Out[81]:  Testov''n''
```

Практическое задание

1. Каждое из слов «разработка», «сокет», «декоратор» представить в строковом формате и проверить тип и содержание соответствующих переменных. Затем с помощью онлайн-конвертера преобразовать строковые представление в формат Unicode и также проверить тип и содержимое переменных.

2. Каждое из слов «class», «function», «method» записать в байтовом типе без преобразования в последовательность кодов (не используя методы **encode** и **decode**) и определить тип, содержимое и длину соответствующих переменных.

3. Определить, какие из слов «attribute», «класс», «функция», «type» невозможно записать в байтовом типе.

4. Преобразовать слова «разработка», «администрирование», «protocol», «standard» из строкового представления в байтовое и выполнить обратное преобразование (используя методы **encode** и **decode**).

5. Выполнить пинг веб-ресурсов yandex.ru, youtube.com и преобразовать результаты из байтовового в строковый тип на кириллице.

6. Создать текстовый файл **test_file.txt**, заполнить его тремя строками: «сетевое программирование», «сокет», «декоратор». Проверить кодировку файла по умолчанию. Принудительно открыть файл в формате Unicode и вывести его содержимое.

Дополнительные материалы

1. Таблица символов Unicode.
2. Юникод-конвертер.
3. Устройство оперативной памяти компьютера.
4. Кодировка символов Unicode.
5. Билл Любанович. Простой Питон. Современный стиль программирования. (Каталог «Дополнительные материалы».)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Python 3 для сетевых инженеров.
2. Unicode: визуализация занятого пространства и объяснение тех аспектов, которые должен знать каждый программист.
3. Кодирование символов. Unicode.
4. Юникод: необходимый практический минимум для каждого разработчика.
5. Юникод для чайников.