Страница файла:
https://info-master.su/programming/python/chto-novogo-v-python-3-2.pdf

Перевод оригинальной документации:
Поляков А.В. (https://vk.com/avprojects)

Что нового в Python 3.2

В Python 3.2 по сравнению с версией 3.1 добавлено довольно много изменений. А именно…

PEP 384: Определение стабильного ABI

В прошлом модули расширения, созданные для одной версии Python, часто не могли использоваться с другими версиями Python. Особенно в Windows, каждый выпуск функций Python требовал перестройки всех модулей расширения, которые хотелось использовать. Это требование было результатом свободного доступа к внутренним компонентам интерпретатора Python, которые могли использовать модули расширения.

Начиная с Python 3.2 становится доступным альтернативный подход: модули расширения, которые ограничивают себя ограниченным API (определённые с Py_LIMITED_API), не могут использовать многие внутренние компоненты, но ограничены набором функций API, которые обещают быть стабильными в течение нескольких выпусков. Как следствие, модули расширения, созданные для 3.2 в этом режиме, также будут работать с 3.3, 3.4 и так далее. Модули расширения, которые используют детали структур памяти, все еще могут быть созданы, но их необходимо будет перекомпилировать для каждого выпуска функций.

PEP 389: Модуль синтаксического анализа командной Строки Argparse

Новый модуль для синтаксического анализа командной строки, `argparse`, был введен для преодоления ограничений `optparse`, которые не обеспечивали поддержку позиционных аргументов (не только опций), подкоманд, требуемых опций и других распространенных шаблонов указания и проверки параметров.

Этот модуль уже имел широкий успех в сообществе в качестве стороннего модуля. Будучи более полнофункциональным, чем его предшественник, модуль `argparse` теперь является предпочтительным модулем для обработки из командной строки. Более старый модуль все еще остается доступным из-за значительного объема устаревшего кода, который зависит от него.

Вот аннотированный пример синтаксического анализатора, показывающий такие функции, как ограничение результатов набором вариантов, указание метаварианта на экране справки, проверка наличия одного или нескольких позиционных аргументов и создание требуемого параметра:

```
import argparse
parser = argparse.ArgumentParser(
      description = 'Manage servers',# main description for help
      epilog = 'Tested on Solaris and Linux') # displayed after help
parser.add_argument('action',                        # argument name
      choices = ['deploy', 'start', 'stop'],  # three allowed values
      help = 'action on each target')         # help msg
parser.add_argument('targets',
      metavar = 'HOSTNAME',             # var name used in help msg
      nargs = '+',                      # require one or more targets
      help = 'url for target machines') # help msg explanation
parser.add_argument('-u', '--user',     # -u or --user option
      required = True,                  # make it a required argument
      help = 'login as user')
```

Пример вызова синтаксического анализатора в командной строке:

```
>>>cmd = 'deploy sneezy.example.com sleepy.example.com -u skycaptain'
>>>result = parser.parse_args(cmd.split())
>>>result.action
'deploy'
>>>result.targets
['sneezy.example.com', 'sleepy.example.com']
>>>result.user
'skycaptain'
```

Пример автоматически сгенерированной справки синтаксического анализатора:

```
>>> parser.parse_args('-h'.split())

usage: manage_cloud.py [-h] -u USER
                       {deploy,start,stop} HOSTNAME [HOSTNAME ...]

Manage servers
```

```
positional arguments:
  {deploy,start,stop}   action on each target
  HOSTNAME              url for target machines

optional arguments:
  -h, --help            show this help message and exit
  -u USER, --user USER  login as user

Tested on Solaris and Linux
```

Особенно приятной функцией `argparse` является возможность определять подразделы, каждый со своими собственными шаблонами аргументов и отображением справки:

```
import argparse
parser = argparse.ArgumentParser(prog='HELM')
subparsers = parser.add_subparsers()

# first subgroup
parser_l = subparsers.add_parser('launch', help='Launch Control')
parser_l.add_argument('-m', '--missiles', action='store_true')
parser_l.add_argument('-t', '--torpedos', action='store_true')

# second subgroup
parser_m = subparsers.add_parser('move', help='Move Vessel',
           aliases=('steer', 'turn'))         # equivalent names
parser_m.add_argument('-c', '--course', type=int, required=True)
parser_m.add_argument('-s', '--speed', type=int, default=0)

$ ./helm.py --help            # top level help (launch and move)
$ ./helm.py launch --help     # help for launch options
$ ./helm.py launch --missiles # set missiles=True and torpedos=False
$ ./helm.py steer --course 180 --speed 5 # set movement parameters
```

PEP 391: Конфигурация на основе словаря для ведения журнала

Модуль ведения журнала (`logging`) предоставлял два вида конфигурации: один стиль с вызовами функций для каждой опции или другой стиль, управляемый внешним файлом, сохраненным в формате `ConfigParser`. Эти параметры не обеспечивали гибкости для создания конфигураций из файлов JSON или YAML, а также не поддерживали инкрементную конфигурацию, которая необходима для указания параметров регистратора из командной строки.

Для поддержки более гибкого стиля модуль теперь предлагает `logging.config.dictConfig()` для указания конфигурации ведения журнала с помощью простых словарей Python. Параметры конфигурации включают в себя форматеры, обработчики, фильтры и регистраторы. Вот рабочий пример словаря конфигурации:

```
{"version": 1,
"formatters":
  {"brief":
  {"format": "%(levelname)-8s: %(name)-15s: %(message)s"},
   "full": {"format":
            "%(asctime)s %(name)-15s %(levelname)-8s %(message)s"} },
"handlers": {"console": {
                    "class": "logging.StreamHandler",
                    "formatter": "brief",
                    "level": "INFO",
                    "stream": "ext://sys.stdout"},
              "console_priority": {
                    "class": "logging.StreamHandler",
                    "formatter": "full",
                    "level": "ERROR",
                    "stream": "ext://sys.stderr"}
              },
"root": {"level": "DEBUG", "handlers":
["console", "console_priority"]}}
```

Если этот словарь хранится в файле с именем `conf.json`, его можно загрузить и вызвать с помощью кода, подобного этому:

```
>>> import json, logging.config
>>> with open('conf.json') as f:
...     conf = json.load(f)
...
>>> logging.config.dictConfig(conf)
>>> logging.info("Transaction completed normally")
INFO    : root           : Transaction completed normally
>>> logging.critical("Abnormal termination")
2011-02-17 11:14:36,694 root           CRITICAL Abnormal termination
```

PEP 3148: Модуль concurrent.futures

Код для создания параллелизма и управления им собирается в новом пространстве имен верхнего уровня `concurrent`. Его первым элементом является пакет `futures`, который обеспечивает единый высокоуровневый интерфейс для управления потоками и процессами.

Дизайн для `concurrent.futures` был вдохновлен пакетом `java.util.concurrent`. В этой модели выполняемый вызов и его результат представлены объектом `Future`, который абстрагирует функции, общие для потоков, процессов и удаленных вызовов процедур. Этот объект поддерживает проверки статуса (запущено или выполнено), тайм-ауты, отмены, добавление обратных вызовов и доступ к результатам или исключениям.

Основным предложением нового модуля является пара классов `executor` для запуска вызовов и управления ими. Цель исполнителей - упростить использование существующих инструментов для выполнения параллельных вызовов. Они экономят усилия, необходимые для настройки пула ресурсов, запуска вызовов, создания очереди результатов, добавления обработки тайм-аута и ограничения общего количества потоков, процессов или вызовов удаленных процедур.

В идеале каждое приложение должно совместно использовать одного исполнителя для нескольких компонентов, чтобы можно было централизованно управлять ограничениями процессов и потоков. Это решает проблему проектирования, возникающую, когда у каждого компонента есть своя собственная конкурирующая стратегия управления ресурсами.

Оба класса имеют общий интерфейс с тремя методами: `submit()` для планирования вызываемого объекта и возврата будущего объекта; `map()` для планирования множества асинхронных вызовов одновременно и `shutdown()` для освобождения ресурсов. Класс является контекстным менеджером (context manager) и может использоваться в операторе `with` для обеспечения автоматического освобождения ресурсов при завершении выполнения ожидающих в данный момент фьючерсов.

Простым примером `ThreadPoolExecutor` является запуск четырех параллельных потоков для копирования файлов:

```
import concurrent.futures, shutil
with concurrent.futures.ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest4.txt')
```

PEP 3147: Каталоги репозитория PYC

Схема Python для кэширования байт-кода в файлах `.pyc` плохо работала в средах с несколькими интерпретаторами Python. Если один интерпретатор столкнется с кэшированным файлом, созданным другим интерпретатором, он перекомпилирует исходный код и перезапишет кэшированный файл, тем самым теряя преимущества кэширования.

Проблема "рус-боев" стала более острой, поскольку дистрибутивы Linux стали обычным явлением для поставки с несколькими версиями Python. Эти конфликты также возникают с альтернативами CPython, такими как Unladen Swallow.

Чтобы решить эту проблему, механизм импорта Python был расширен, чтобы использовать разные имена файлов для каждого интерпретатора. Вместо Python 3.2 и Python 3.3 и Unladen Swallow, конкурирующих за файл с именем `"mymodule.pyc"`, теперь они будут искать `"mymodule.cpython-32.pyc"`, `"mymodule.cpython-33.pyc"` и `"mymodule.порожний 10.pyc"`. И чтобы все эти новые файлы не загромождали исходные каталоги, файлы `pyc` теперь собраны в каталоге `"__pycache__"`, хранящемся в каталоге пакетов.

Помимо имен файлов и целевых каталогов, новая схема имеет несколько аспектов, которые видны программисту:

- Импортированные модули теперь имеют атрибут `__cached__`, в котором хранится имя фактического файла, который был импортирован:

```
>>> import collections
>>> collections.__cached__
'c:/py32/lib/__pycache__/collections.cpython-32.pyc'
```

- Тег, уникальный для каждого интерпретатора, доступен из модуля `imp`:

```
>>> import imp
>>> imp.get_tag()
'cpython-32'
```

Скрипты, которые пытаются вывести исходное имя файла из импортированного файла, теперь должны быть умнее. Больше недостаточно просто удалить букву `"c"` из имени файла `".pyc"`. Вместо этого используйте новые функции в модуле `imp`:

```
>>>imp.source_from_cache(
  'c:/py32/lib/__pycache__/collections.cpython-32.pyc')
'c:/py32/lib/collections.py'
>>>imp.cache_from_source('c:/py32/lib/collections.py')
```

```
'c:/py32/lib/__pycache__/collections.cpython-32.pyc'
```

Модули `py_compile` и `compile all` были обновлены, чтобы отразить новое соглашение об именовании и целевой каталог. Вызов командной строки `compileall` имеет новые опции: `-i` для указания списка файлов и каталогов для компиляции и `-b`, который приводит к записи файлов байт-кода в их устаревшее местоположение, а не в `__pycache__`.

Модуль `importlib.abc` был обновлен новыми абстрактными базовыми классами для загрузки файлов байт-кода. Устаревшие ABCS, Payloader и PyPycLoader, были признаны устаревшими (инструкции о том, как поддерживать совместимость с Python 3.1, включены в документацию).

PEP 3149: Файлы версии ABI с тегом .so

Каталог репозитория PYC позволяет совместно размещать несколько файлов кэша байт-кода. Этот PEP реализует аналогичный механизм для общих объектных файлов, предоставляя им общий каталог и разные имена для каждой версии.

Общим каталогом является "`pyshared`", а имена файлов различаются путем указания реализации Python (например, CPython, PyPy, Jython и т.д.), номеров основной и второстепенной версий и необязательных флагов сборки (таких как "`d`" для `debug`, "`m`" для `pymalloc`, "`u`" для широкого использования в юникоде). Для произвольного пакета "`foo`" вы можете увидеть эти файлы при установке дистрибутива:

```
/usr/share/pyshared/foo.cpython-32m.so
/usr/share/pyshared/foo.cpython-33md.so
```

В самом Python теги доступны из функций в модуле `sysconfig`:

```
>>> import sysconfig
# find the version tag
>>> sysconfig.get_config_var('SOABI')
'cpython-32mu'
# find the full filename extension
>>> sysconfig.get_config_var('EXT_SUFFIX')
'.cpython-32mu.so'
```

PEP 3333: Интерфейс шлюза веб-сервера Python v1.0.1

Этот информационный PEP разъясняет, как проблемы с байтами/текстом должны обрабатываться протоколом WSGI. Проблема заключается в том, что обработка строк в Python 3 наиболее удобно обрабатывается с помощью типа `str`, даже несмотря на то, что сам протокол HTTP ориентирован на байты.

PEP отличает так называемые собственные строки (native strings), которые используются для заголовков запросов/ответов и метаданных, от байтовых строк, которые используются для тела запросов и ответов.

Собственные строки всегда имеют тип `str`, но ограничены кодовыми точками от U +0000 до U +00FF, которые можно перевести в байты с использованием кодировки Latin-1. Эти строки используются для ключей и значений в словаре среды, а также для заголовков и статусов ответов в функции `start_response()`. Они должны следовать RFC 2616 в отношении кодирования. То есть они должны быть либо символами ISO-8859-1, либо использовать MIME-кодировку RFC 2047.

Для разработчиков, переносящих приложения WSGI с Python 2, вот основные моменты:

- Если приложение уже использовало строки для заголовков в Python 2, никаких изменений не требуется.
- Если вместо этого приложение кодировало выходные заголовки или декодировало входные заголовки, то заголовки необходимо будет перекодировать в Latin-1. Например, выходной заголовок, закодированный в utf-8, был с использованием `h.encode('utf-8')`, теперь необходимо преобразовать из байтов в собственные строки, используя `h.encode('utf-8').decode('latin-1')`.
- Значения, полученные приложением или отправленные с помощью метода `write()`, должны быть байтовыми строками. Функция `start_response()` и окружающая среда должны использовать собственные строки. Эти два понятия нельзя смешивать.

Для разработчиков серверов, пишущих пути CGI-to-WSGI или другие протоколы в стиле CGI, пользователи должны иметь возможность доступа к среде с использованием собственных строк, даже если базовая платформа может иметь другое соглашение. Чтобы устранить этот пробел, модуль `wsgiref` имеет новую функцию `wsgiref.handlers.read_environ()` для перекодирования переменных CGI из `os.environ` в собственные строки и возврата нового словаря.

Другие языковые изменения

Некоторые небольшие изменения, внесенные в основной язык Python, заключаются в следующем:

Форматирование строк для `format()` и `str.format()` получило новые возможности для символа формата #. Ранее для целых чисел в двоичном, восьмеричном или шестнадцатеричном формате это приводило к тому, что выходные данные имели префикс '`0b`', '`0o`' или '`0x`' соответственно. Теперь он также может обрабатывать числа с плавающей запятой, сложные и десятичные, в результате чего выходные данные всегда будут содержать десятичную точку, даже если за ней не следует никаких цифр.

```
>>> format(20, '#o')
'0o24'
>>> format(12.34, '#5.0f')
'  12.'
```

Существует также новый метод `str.format_map()`, который расширяет возможности существующего метода `str.format()`, принимая произвольные объекты отображения. Этот новый метод позволяет использовать форматирование строк с любым из множества объектов, похожих на словарь Python, таких как `defaultdict`, `Shelf`, `ConfigParser` или `dbm`. Это также полезно с пользовательскими подклассами `dict`, которые нормализуют ключи перед поиском или которые предоставляют метод `__missing__()` для неизвестных ключей:

```
>>> import shelve
>>> d = shelve.open('tmp.shl')
>>> 'The {project_name} status is {status} as of
{date}'.format_map(d)
'The testing project status is green as of February 15, 2011'

>>> class LowerCasedDict(dict):
...     def __getitem__(self, key):
...         return dict.__getitem__(self, key.lower())
>>> lcd = LowerCasedDict(part='widgets', quantity=10)
>>> 'There are {QUANTITY} {Part} in stock'.format_map(lcd)
'There are 10 widgets in stock'

>>> class PlaceholderDict(dict):
...     def __missing__(self, key):
...         return '<{}>'.format(key)
>>> 'Hello {name}, welcome to
{location}'.format_map(PlaceholderDict())
'Hello <name>, welcome to <location>'
```

Интерпретатор теперь можно запустить с помощью тихой опции **`-q`**, чтобы информация об авторских правах и версии не отображалась в интерактивном режиме. Параметр может быть проанализирован с помощью атрибута `sys.flags`:

```
$ python -q
>>> sys.flags
sys.flags(debug=0, division_warning=0, inspect=0, interactive=0,
optimize=0, dont_write_bytecode=0, no_user_site=0, no_site=0,
ignore_environment=0, verbose=0, bytes_warning=0, quiet=1)
```

Функция `hasattr()` работает путем вызова `getattr()` и определения того, вызвано ли исключение. Этот метод позволяет обнаруживать методы, созданные динамически с помощью `__getattr__()` или `__getattribute__()`, которые в противном случае отсутствовали бы в словаре классов. Раньше `hasattr` перехватывал любое исключение, возможно, маскируя подлинные ошибки. Теперь `hasattr` был ужесточен, чтобы перехватывать только `AttributeError` и пропускать другие исключения:

```
>>> class A:
...     @property
...     def f(self):
...         return 1 // 0
...
>>> a = A()
>>> hasattr(a, 'f')
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

`str()` числа с плавающей точкой или комплексного числа теперь такая же, как и его `repr()`. Ранее форма `str()` была короче, но это просто вызывало путаницу и больше не требуется теперь, когда по умолчанию отображается кратчайший возможный `repr()`:

```
>>> import math
>>> repr(math.pi)
'3.141592653589793'
>>> str(math.pi)
'3.141592653589793'
```

Объекты `memoryview` теперь имеют метод `release()`, и они также теперь поддерживают протокол управления контекстом. Это позволяет своевременно освобождать любые ресурсы, которые были получены при запросе буфера у исходного объекта.

```
>>> with memoryview(b'abcdefgh') as v:
...     print(v.tolist())
[97, 98, 99, 100, 101, 102, 103, 104]
```

Ранее было запрещено удалять имя из локального пространства имен, если оно встречается как свободная переменная во вложенном блоке:

```
def outer(x):
    def inner():
        return x
    inner()
    del x
```

Это недопустимо. Помните, что цель предложения `except` очищена, поэтому этот код, который раньше работал с Python 2.6, вызвал ошибку `SyntaxError` в Python 3.1, а теперь работает снова:

```
def f():
    def print_error():
        print(e)
    try:
        something
    except Exception as e:
        print_error()
        # implicit "del e" here
```

Инструмент внутренней `structsequence` теперь создает подклассы `tuple`. Это означает, что структуры C, подобные тем, которые возвращаются `os.stat()`, `time.gmtime()` и `sys.version_info`, теперь работают как именованный кортеж и теперь работают с функциями и методами, которые ожидают кортеж в качестве аргумента. Это большой шаг вперед в том, чтобы сделать структуры C такими же гибкими, как и их чистые аналоги на Python:

```
>>> import sys
>>> isinstance(sys.version_info, tuple)
True
>>> 'Version %d.%d.%d %s(%d)' % sys.version_info
'Version 3.2.0 final(0)'
```

Предупреждениями теперь легче управлять, используя переменную среды PYTHONWARNINGS в качестве альтернативы использованию `-W` в командной строке:

```
$ export
PYTHONWARNINGS='ignore::RuntimeWarning::,once::UnicodeWarning::'
```

Добавлена новая категория предупреждений - `ResourceWarning`. Она выдается при обнаружении потенциальных проблем с потреблением ресурсов или очисткой. Она отключена по умолчанию в обычных сборках выпуска, но может быть включена с помощью средств, предоставляемых модулем предупреждений, или в командной строке.

`ResourceWarning` выдается при завершении работы интерпретатора, если список `gc.garbage` не пуст, и если установлен параметр `gc.DEBUG_UNCOLLECTABLE`, печатаются все объекты, которые невозможно собрать. Это предназначено для того, чтобы программист знал, что их код содержит проблемы с завершением объекта.

`ResourceWarning` также выдается, когда файловый объект уничтожается без явного закрытия. В то время как средство освобождения для такого объекта гарантирует, что он закрывает базовый ресурс операционной системы (обычно, файловый дескриптор), задержка в освобождении объекта может вызвать различные проблемы, особенно в Windows. Вот пример включения предупреждения из командной строки:

```
$ python -q -Wdefault
>>> f = open("foo", "wb")
>>> del f
__main__:1: ResourceWarning: unclosed file <_io.BufferedWriter
name='foo'>
```

Объекты `range` теперь поддерживают методы `index` и `count`. Это часть усилий по созданию большего количества объектов, полностью реализующих `collections.Sequence` абстрактного базового класса. В результате язык будет иметь более единообразный API. Кроме того, объекты `range` теперь поддерживают нарезку и отрицательные индексы, даже со значениями, превышающими `sys.maxsize`. Это делает `range` более совместимым со списками:

```
>>> range(0, 100, 2).count(10)
1
>>> range(0, 100, 2).index(10)
5
>>> range(0, 100, 2)[5]
10
>>> range(0, 100, 2)[0:5]
range(0, 10, 2)
```

Встроенная функция `callable()` из Py2.x была восстановлена. Это обеспечивает краткую, читаемую альтернативу использованию абстрактного базового класса в выражении типа `isinstance(x, collections.Callable)`:

```
>>> callable(max)
True
>>> callable(20)
False
```

Механизм импорта Python теперь может загружать модули, установленные в каталогах с символами в имени пути, отличными от ASCII. Это решило усугубляющуюся проблему с домашними каталогами для пользователей с символами, отличными от ASCII, в их именах пользователей.

Новые, улучшенные и устаревшие модули

Стандартная библиотека Python претерпела значительные изменения по техническому обслуживанию и улучшению качества.

Самая большая новость для Python 3.2 заключается в том, что модули `mailbox`, `module` и `nntplib` теперь корректно работают с моделью байт/текст в Python 3. Впервые обеспечивается корректная обработка сообщений со смешанными кодировками.

Во всей стандартной библиотеке более пристальное внимание уделялось кодировкам и проблемам соотношения текста и байтов. В частности, взаимодействие с операционной системой теперь позволяет лучше обмениваться данными, отличными от ASCII, используя кодировку Windows MBCS, кодировки с учетом локали или UTF-8.

Еще одним значительным преимуществом является добавление существенно улучшенной поддержки SSL-соединений и сертификатов безопасности.

Кроме того, в большем количестве классов теперь реализован контекстный менеджер (context manager) для поддержки удобной и надежной очистки ресурсов с помощью инструкции `with`.

email

Удобство использования пакета `email` в Python 3 было в основном налажено благодаря обширным усилиям Р. Дэвида Мюррея. Проблема заключалась в том, что электронные письма обычно считываются и хранятся в виде байтов (`bytes`), а не в виде текста (как `str`), и они могут содержать несколько кодировок в одном электронном письме. Таким образом, пакет электронной почты пришлось расширить, чтобы анализировать и генерировать сообщения электронной почты в формате байтов.

Новые функции `message_from_bytes()` и `message_from_binary_file()`, а также новые классы `BytesFeedParser` и `BytesParser` позволяют анализировать двоичные данные сообщений в объектную модель.

Учитывая байты, введенные в модель, `get_payload()` по умолчанию декодирует тело сообщения, которое имеет 8-битную кодировку для передачи содержимого, используя кодировку, указанную в заголовках MIME, и возвращает результирующую строку.

Учитывая байты, введенные в модель, генератор преобразует тела сообщений, которые имеют 8-битную кодировку для передачи содержимого, в 7-битную кодировку для передачи содержимого.

Заголовки с некодированными байтами, отличными от ASCII, считаются закодированными в RFC 2047 с использованием неизвестного 8-битного набора символов.

Новый класс `BytesGenerator` выдает байты в качестве выходных данных, сохраняя любые неизменные данные, отличные от ASCII, которые присутствовали во входных данных, используемых для построения модели, включая тела сообщений с 8-битной кодировкой передачи содержимого.

The smtplib SMTP class now accepts a byte string for the msg argument to the sendmail() method, and a new method, send_message() accepts a Message object and can optionally obtain the from_addr and to_addrs addresses directly from the object.
Класс `smtplib SMTP` теперь принимает байтовую строку в качестве аргумента `msg` для метода `sendmail()`, а новый метод `send_message()` принимает объект `Message` и может дополнительно получать адреса `from_addr` и `to_addr` непосредственно из объекта.

elementtree

Пакет `xml.etree.ElementTree` и его аналог `xml.etree.cElementTree` был обновлен до версии 1.3. Было добавлено несколько новых и полезных функций и методов:

- `xml.etree.ElementTree.fromstringlist()`, который создает XML-документ из последовательности фрагментов
- `xml.etree.ElementTree.register_namespace()` для регистрации глобального префикса пространства имен
- `xml.etree.ElementTree.tostringlist()` для строкового представления, включающего все подсписки
- `xml.etree.ElementTree.Element.extend()` для добавления последовательности из нуля или более элементов
- `xml.etree.ElementTree.Element.iterfind()` выполняет поиск элемента и подэлементов
- `xml.etree.ElementTree.Element.itertext()` создает текстовый итератор над элементом и его подэлементами
- `xml.etree.ElementTree.TreeBuilder.end()` закрывает текущий элемент
- `xml.etree.ElementTree.TreeBuilder.doctype()` обрабатывает объявление `doctype`

Два метода были признаны устаревшими:

- `xml.etree.ElementTree.getchildren()` - вместо этого используйте `list(elem)`
- `xml.etree.ElementTree.getiterator()` - вместо этого используйте `Element.iter`.

functools

Модуль `functools` включает в себя новый декоратор для кэширования вызовов функций. `functools.lru_cache()` может сохранять повторяющиеся запросы на внешний ресурс всякий раз, когда ожидается, что результаты будут одинаковыми.

Например, добавление декоратора кэширования к функции запроса к базе данных может сэкономить доступ к базе данных для популярных поисковых запросов:

```
>>> import functools
>>> @functools.lru_cache(maxsize=300)
... def get_phone_number(name):
...     c = conn.cursor()
...     c.execute('SELECT phonenumber FROM phonelist WHERE name=?',
(name,))
```

```
...     return c.fetchone()[0]

>>> for name in user_requests:
...     get_phone_number(name)        # cached lookup
```

Чтобы помочь с выбором эффективного размера кэша, функция `wrapped` предназначена для отслеживания статистики кэша:

```
>>> get_phone_number.cache_info()
CacheInfo(hits=4805, misses=980, maxsize=300, currsize=300)
```

Если таблица списка телефонов обновляется, устаревшее содержимое кэша может быть очищено с помощью:

```
>>> get_phone_number.cache_clear()
```

`functools.wraps()` теперь добавляет атрибут `__wrapped__`, указывающий на исходную вызываемую функцию. Это позволяет проводить самоанализ обернутых функций. Он также копирует `__annotations__`, если они определены. И теперь он также изящно пропускает отсутствующие атрибуты, такие как `__doc__`, которые могут быть не определены для обернутого вызываемого объекта.

В приведенном выше примере кэш может быть удален путем восстановления исходной функции:

```
>>> get_phone_number = get_phone_number.__wrapped__     # uncached
function
```

Чтобы помочь писать классы с расширенными методами сравнения, новый декоратор `functools.total_ordering()` будет использовать существующие методы равенства и неравенства для заполнения оставшихся методов.

Например, предоставление `__eq__` и `__lt__` позволит `total_ordering()` заполнить `__le__`, `__gt__` и `__ge__`:

```
@total_ordering
class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))

    def __lt__(self, other):
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

С помощью декоратора `total_ordering` остальные методы сравнения заполняются автоматически.

Чтобы облегчить перенос программ с Python 2, функция `functools.cmp_to_key()` преобразует функцию сравнения старого стиля в современную функцию ключа (сортировки):

```
>>> # locale-aware sort order
>>> sorted(iterable, key=cmp_to_key(locale.strcoll))
```

itertools

Модуль `itertools` имеет новую функцию `accumulate()`, созданную по образцу оператора сканирования APL и функции накопления `Numpy`:

```
>>> from itertools import accumulate
>>> list(accumulate([8, 2, 50]))
[8, 10, 60]
>>> prob_dist = [0.1, 0.4, 0.2, 0.3]
>>> list(accumulate(prob_dist))      # cumulative probability
distribution
[0.1, 0.5, 0.7, 1.0]
```

Для примера использования `accumulate()` смотрите примеры для модуля `random`.

collections

Класс `collections.Counter` теперь имеет две формы вычитания на месте, существующий оператор `-=` для насыщающего вычитания (saturating subtraction) и новый метод `subtract()` для обычного вычитания. Первый подходит для мультимножеств, которые имеют только положительные счётчики, а второй больше подходит для вариантов использования, допускающих отрицательные счётчики:

```
>>> from collections import Counter
>>> tally = Counter(dogs=5, cats=3)
>>> tally -= Counter(dogs=2, cats=8)    # saturating subtraction
>>> tally
Counter({'dogs': 3})

>>> tally = Counter(dogs=5, cats=3)
>>> tally.subtract(dogs=2, cats=8)      # regular subtraction
>>> tally
Counter({'dogs': 3, 'cats': -5})
```

Класс `collections.OrderedDict` имеет новый метод `move_to_end()`, который принимает существующий ключ и перемещает его либо в первую, либо в последнюю позицию в упорядоченной последовательности.

По умолчанию элемент перемещается на последнюю позицию. Это эквивалентно обновлению записи с помощью `od[k] = od.pop(k)`.

Быстрая операция перехода к концу полезна для изменения последовательности записей. Например, упорядоченный словарь можно использовать для отслеживания порядка доступа по возрастанию записей от самых старых до самых последних, к которым обращались.

```
>>> from collections import OrderedDict
>>> d = OrderedDict.fromkeys(['a', 'b', 'X', 'd', 'e'])
>>> list(d)
['a', 'b', 'X', 'd', 'e']
>>> d.move_to_end('X')
>>> list(d)
['a', 'b', 'd', 'e', 'X']
```

Класс `collections.deque` расширил два новых метода `count()` и `reverse()`, которые делают их более заменяемыми для объектов списка:

```
>>> from collections import deque
>>> d = deque('simsalabim')
>>> d.count('s')
2
>>> d.reverse()
>>> d
deque(['m', 'i', 'b', 'a', 'l', 'a', 's', 'm', 'i', 's'])
```

threading

Модуль `threading` имеет новый класс синхронизации `Barrier` для того, чтобы заставить несколько потоков ждать, пока все они не достигнут общей точки барьера. Барьеры полезны для обеспечения того, чтобы задача с несколькими предварительными условиями не выполнялась до тех пор, пока не будут завершены все предшествующие задачи.

Барьеры могут работать с произвольным количеством потоков. Это обобщение `Rendezvous`, которое определено только для двух потоков.

Реализованные в виде двухфазного циклического барьера, объекты `Barrier` подходят для использования в циклах. Отдельные фазы наполнения и слива гарантируют, что все потоки высвобождаются (сливаются) до того, как какой-либо из них сможет вернуться в цикл и снова войти в барьер. Барьер полностью сбрасывается после каждого цикла.

Пример использования:

```
from threading import Barrier, Thread

def get_votes(site):
    ballots = conduct_election(site)
    all_polls_closed.wait()        # do not count until all polls are
closed
    totals = summarize(ballots)
    publish(site, totals)

all_polls_closed = Barrier(len(sites))
for site in sites:
    Thread(target=get_votes, args=(site,)).start()
```

В этом примере барьер обеспечивает соблюдение правила, согласно которому голоса не могут быть подсчитаны ни на одном избирательном участке до тех пор, пока все участки не будут закрыты. Обратите внимание, как решение с барьером похоже на решение с `threading.Thread.join()`, но потоки остаются живыми и продолжают выполнять работу (суммирование бюллетеней) после пересечения точки барьера.

Если какая-либо из предыдущих задач может зависнуть или быть отложена, можно создать барьер с необязательным параметром тайм-аута (`timeout`). Затем, если период ожидания истекает до того, как все предшествующие задачи достигнут точки барьера, все ожидающие потоки освобождаются и возникает исключение `BrokenBarrierError`:

```
def get_votes(site):
    ballots = conduct_election(site)
    try:
        all_polls_closed.wait(timeout=midnight - time.now())
    except BrokenBarrierError:
        lockbox = seal_ballots(ballots)
        queue.put(lockbox)
    else:
        totals = summarize(ballots)
        publish(site, totals)
```

В этом примере барьер применяет более надежное правило. Если некоторые избирательные участки не заканчивают работу до полуночи, время ожидания барьера истекает, а бюллетени опечатываются и помещаются в очередь для последующей обработки.

Смотрите Barrier Synchronization Patterns (Шаблоны синхронизации барьеров) для получения дополнительных примеров того, как барьеры могут использоваться в параллельных вычислениях. Кроме того, в Небольшой книге о семафорах (Little Book of Semaphores), раздел 3.6, есть простое, но подробное объяснение барьеров.

datetime и time

Модуль `datetime` имеет новый тип `timezone`, который реализует интерфейс `tzinfo`, возвращая фиксированное смещение UTC и имя часового пояса. Это упрощает создание объектов `datetime` с учетом часового пояса:

```
>>> from datetime import datetime, timezone

>>> datetime.now(timezone.utc)
datetime.datetime(2010, 12, 8, 21, 4, 2, 923754,
tzinfo=datetime.timezone.utc)

>>> datetime.strptime("01/01/2000 12:00 +0000", "%m/%d/%Y %H:%M %z")
datetime.datetime(2000, 1, 1, 12, 0, tzinfo=datetime.timezone.utc)
```

Кроме того, объекты `timedelta` теперь можно умножать на `float` и делить на объекты `float` и `int`. И объекты `timedelta` теперь могут разделять друг друга.

Метод `datetime.date.strftime()` больше не ограничен годами после 1900 года. Новый поддерживаемый диапазон лет составляет от 1000 до 9999 включительно.

Всякий раз, когда в кортеже времени используется двузначный год, интерпретация регулируется `time.accept2dyear`. Значение по умолчанию равно `True`, что означает, что для двузначного года столетие угадывается в соответствии с правилами POSIX, регулирующими формат `%y strptime`.

Начиная с Py3.2, использование эвристики угадывания века будет выдавать предупреждение об устаревании (`DeprecationWarning`).

Вместо этого рекомендуется установить значение `time.accept2dyear` равным `False`, чтобы можно было использовать большие диапазоны дат без догадок:

```
>>> import time, warnings
>>> warnings.resetwarnings()       # remove the default warning
filters

>>> time.accept2dyear = True       # guess whether 11 means 11 or 2011
>>> time.asctime((11, 1, 1, 12, 34, 56, 4, 1, 0))
Warning (from warnings module):
  ...
DeprecationWarning: Century info guessed for a 2-digit year.
'Fri Jan  1 12:34:56 2011'

>>> time.accept2dyear = False      # use the full range of allowable
dates
>>> time.asctime((11, 1, 1, 12, 34, 56, 4, 1, 0))
'Fri Jan  1 12:34:56 11'
```

Некоторые функции теперь имеют значительно расширенные диапазоны дат. Когда значение `time.accept2dyear` равно `false`, функция `time.asctime()` будет принимать любой год, который вписывается в C `int`, в то время как функции `time.mktime()` и `time.strftime()` будут принимать полный диапазон, поддерживаемый соответствующими функциями операционной системы.

math

Модуль `math` был обновлен шестью новыми функциями, вдохновленными стандартом C99.

Функция `isfinite()` обеспечивает надежный и быстрый способ определения специальных значений. Она возвращает значение `True` для обычных чисел и значение `False` для `Nan` или `Infinity`:

```
>>> from math import isfinite
>>> [isfinite(x) for x in (123, 4.56, float('Nan'), float('Inf'))]
[True, True, False, False]
```

Функция `expm1()` вычисляет `e ** x-1` для малых значений `x` без потери точности, которое обычно сопровождает вычитание почти равных величин:

```
>>> from math import expm1
>>> expm1(0.013671875)   # more accurate way to compute e**x-1 for a
small x
0.013765762467652909
```

Функция `erf()` вычисляет интеграл вероятности или гауссову функцию ошибки. Дополнительная функция ошибки, `erfc()`, равна `1 - erf(x)`:

```
>>> from math import erf, erfc, sqrt
>>> erf(1.0/sqrt(2.0))   # portion of normal distribution within 1
standard deviation
0.682689492137086
>>> erfc(1.0/sqrt(2.0))  # portion of normal distribution outside 1
standard deviation
0.31731050786291404
>>> erf(1.0/sqrt(2.0)) + erfc(1.0/sqrt(2.0))
1.0
```

Функция `gamma()` является непрерывным расширением факториальной функции. Поскольку функция связана с факториалами, она становится большой даже при малых значениях `x`, поэтому существует также функция `gamma()` для вычисления натурального логарифма гамма-функции:

```
>>> from math import gamma, lgamma
>>> gamma(7.0)           # six factorial
720.0
>>> lgamma(801.0)        # log(800 factorial)
4551.950730698041
```

abc

Модуль `abc` теперь поддерживает методы `abstractclassmethod()` и `abstractstaticmethod()`.

Эти инструменты позволяют определить абстрактный базовый класс, для реализации которого требуется определенный `classmethod()` или `staticmethod()`:

```
class Temperature(metaclass=abc.ABCMeta):
    @abc.abstractclassmethod
    def from_fahrenheit(cls, t):
        ...
    @abc.abstractclassmethod
    def from_celsius(cls, t):
        ...
```

io

В `io.BytesIO` появился новый метод `getbuffer()`, который обеспечивает функциональность, аналогичную `memoryview()`. Это создает редактируемое представление данных без создания копии. Произвольный доступ к буферу и поддержка нотации срезов хорошо подходят для редактирования на месте:

```
>>> REC_LEN, LOC_START, LOC_LEN = 34, 7, 11

>>> def change_location(buffer, record_number, location):
...     start = record_number * REC_LEN + LOC_START
...     buffer[start: start+LOC_LEN] = location

>>> import io

>>> byte_stream = io.BytesIO(
...     b'G3805  storeroom  Main chassis    '
...     b'X7899  shipping   Reserve cog     '
...     b'L6988  receiving  Primary sprocket'
... )
>>> buffer = byte_stream.getbuffer()
>>> change_location(buffer, 1, b'warehouse  ')
>>> change_location(buffer, 0, b'showroom   ')
>>> print(byte_stream.getvalue())
b'G3805  showroom   Main chassis    '
b'X7899  warehouse  Reserve cog     '
b'L6988  receiving  Primary sprocket'
```

reprlib

При написании метода `__repr__()` для пользовательского контейнера легко забыть обработать случай, когда элемент ссылается обратно на сам контейнер. Встроенные объекты Python, такие как `list` и `set`, обрабатывают ссылку на самого себя, отображая "`...`" в рекурсивной части строки представления.

Чтобы помочь написать такие методы `__repr__()`, модуль `reprlib` имеет новый декоратор `recursive_repr()` для обнаружения рекурсивных вызовов `__repr__()` и замены строки-заполнителя вместо этого:

```
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
```

```
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

logging

В дополнение к конфигурации на основе словаря, описанной выше, пакет ведения журнала `logging` имеет множество других улучшений.

Документация по ведению журнала была дополнена базовым руководством, расширенным руководством и книгой рецептов ведения журнала. Эти документы - самый быстрый способ узнать о ведении журнала.

Встроенная функция `logging.basicConfig()` получила аргумент стиля (`style`) для поддержки трех различных типов форматирования строк. По умолчанию используется значение "%" для традиционного форматирования %, может быть установлено значение "{" для нового стиля `str.format()` или может быть установлено значение "$" для форматирования в стиле оболочки, предоставляемого `string.Template`. Следующие три конфигурации эквивалентны:

```
>>> from logging import basicConfig
>>> basicConfig(style='%', format="%(name)s -> %(levelname)s:
%(message)s")
>>> basicConfig(style='{', format="{name} -> {levelname} {message}")
>>> basicConfig(style='$', format="$name -> $levelname: $message")
```

Если до возникновения события протоколирования конфигурация не настроена, теперь существует конфигурация по умолчанию, использующая `StreamHandler`, направленный в `sys.stderr` для событий уровня ПРЕДУПРЕЖДЕНИЯ (WARNING) или выше. Раньше событие, происходящее до настройки конфигурации, либо вызывало исключение, либо автоматически отбрасывало событие в зависимости от значения `logging.raiseExceptions`. Новый обработчик по умолчанию сохраняется в `logging.lastResort`.

Использование фильтров было упрощено. Вместо создания объекта `Filter` предикатом может быть любой вызываемый объект Python, который возвращает True или False.

Был внесен ряд других улучшений, которые добавляют гибкости и упрощают настройку. Смотрите документацию по модулю для получения полного списка изменений в Python 3.2.

csv

Модуль `csv` теперь поддерживает новый диалект `unix_dialect`, который применяет кавычки для всех полей и традиционный стиль Unix с `'\n'` в качестве окончания строки. Зарегистрированное имя диалекта - `unix`.

В `csv.DictWriter` есть новый метод `writeheader()` для записи начальной строки для документирования имен полей:

```
>>> import csv, sys
>>> w = csv.DictWriter(sys.stdout, ['name', 'dept'], dialect='unix')
>>> w.writeheader()
"name","dept"
>>> w.writerows([
...     {'name': 'tom', 'dept': 'accounting'},
...     {'name': 'susan', 'dept': 'Salesl'}])
"tom","accounting"
"susan","sales"
```

contextlib

Существует новый и немного умопомрачительный инструмент `ContextDecorator`, который полезен для создания контекстного менеджера (context manager), выполняющего двойную функцию декоратора функций.

Для удобства эта новая функциональность используется с `contextmanager()`, так что для поддержки обеих ролей не требуется никаких дополнительных усилий.

Основная идея заключается в том, что как контекстные менеджеры, так и декораторы функций могут использоваться для оболочек до и после действия. Менеджеры контекста оборачивают группу операторов с помощью оператора `with`, а декораторы функций оборачивают группу операторов, заключенных в функцию. Таким образом, иногда возникает необходимость написать оболочку до или после действия, которую можно использовать в любой роли.

Например, иногда бывает полезно обернуть функции или группы операторов с помощью регистратора, который может отслеживать время входа и время выхода. Вместо того, чтобы писать для задачи декоратор функций и диспетчер контекста, `contextmanager()` предоставляет обе возможности в одном определении:

```
from contextlib import contextmanager
import logging

logging.basicConfig(level=logging.INFO)
```

```
@contextmanager
def track_entry_and_exit(name):
    logging.info('Entering: %s', name)
    yield
    logging.info('Exiting: %s', name)
```

Раньше это можно было использовать только в качестве контекстного менеджера:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

Теперь его можно использовать и в качестве декоратора:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Попытка выполнять две роли одновременно накладывает некоторые ограничения на технику. Контекстные менеджеры обычно обладают гибкостью для возврата аргумента, используемого оператором `with`, но для декораторов функций нет параллели.

В приведенном выше примере не существует чистого способа для контекстного менеджера `track_entry_and_exit` возвращать экземпляр журнала для использования в теле вложенных инструкций.

decimal и fractions

Марк Дикинсон разработал элегантную и эффективную схему для обеспечения того, чтобы разные числовые типы данных имели одинаковое хэш-значение всякий раз, когда их фактические значения равны:

```
assert hash(Fraction(3, 2)) == hash(1.5) == \
       hash(Decimal("1.5")) == hash(complex(1.5, 0))
```

Некоторые детали хеширования отображаются с помощью нового атрибута `sys.hash_info`, который описывает разрядность хеш-значения, модуль простого числа, значения хеширования для бесконечности (`infinity`) и `nan`, а также множитель, используемый для мнимой части числа:

```
>>> sys.hash_info
sys.hash_info(width=64, modulus=2305843009213693951, inf=314159,
nan=0, imag=1000003)
```

Ранее принятое решение об ограничении совместимости различных числовых типов было смягчено. По-прежнему не поддерживается (и не рекомендуется) неявное смешивание в арифметических выражениях, таких как `Decimal('1.1') + float('1.1')`, поскольку последнее теряет информацию в процессе построения двоичного числа с плавающей точкой. Однако, поскольку существующее значение с плавающей запятой может быть преобразовано без потерь в десятичное или рациональное представление, имеет смысл добавить их в конструктор и поддерживать сравнения смешанного типа.

- Конструктор `decimal.Decimal` теперь принимает объекты с плавающей запятой напрямую, поэтому больше нет необходимости использовать метод `from_float()`.
- Fraction (bpo-2531 and bpo-8188).Сравнения смешанных типов теперь полностью поддерживаются, так что объекты `Decimal` можно напрямую сравнивать с объектами `float` и `fractions.Fraction`.

Аналогичные изменения были внесены и в `fractions.Fraction`, так что методы `from_float()` и `from_decimal()` больше не нужны:

```
>>> from decimal import Decimal
>>> from fractions import Fraction
>>> Decimal(1.1)
Decimal('1.100000000000000088817841970012523233890533447265625')
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
```

Еще одним полезным изменением для модуля `decimal` является то, что атрибут `Context.clamp` теперь является общедоступным. Это полезно при создании контекстов, соответствующих форматам десятичного обмена, указанным в IEEE 754.

ftp

Класс `ftplib.FTP` теперь поддерживает протокол управления контекстом для безоговорочного использования исключений `socket.error` и закрытия FTP-соединения по завершении:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
        ftp.login()
        ftp.dir()

'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
```

Другие файлоподобные объекты, такие как `mmap.mmap` и `file input.input()`, также стали автоматически закрывающимися контекстными менеджерами:

```
with fileinput.input(files=('log1.txt', 'log2.txt')) as f:
    for line in f:
        process(line)
```

Класс `FTP_TLS` теперь принимает параметр контекста, который представляет собой объект `ssl.SSLContext`, позволяющий объединять параметры конфигурации SSL, сертификаты и закрытые ключи в единую (потенциально долговечную) структуру.

popen

Функции `Os.popen()` и `subprocess.Popen()` теперь поддерживаются с помощью инструкций для автоматического закрытия файловых дескрипторов.

select

Модуль `select` теперь предоставляет новый постоянный атрибут `PIPE_BUF`, который дает минимальное количество байтов, которые гарантированно не блокируются, когда `select.select()` сообщает, что канал готов к записи.

```
>>> import select
>>> select.PIPE_BUF
512
```

gzip и zipfile

`gzip.GzipFile` теперь реализует абстрактный базовый класс `io.BufferedIOBase` (за исключением `truncate()`). Он также имеет метод `peek()` и поддерживает невидимые, а также файловые объекты с нулевым заполнением.

Модуль `gzip` также получает функции `compress()` и `decompress()` для упрощения сжатия и распаковки в памяти. Имейте в виду, что текст должен быть закодирован в байтах перед сжатием и распаковкой:

```
>>> import gzip
>>> s = 'Three shall be the number thou shalt count, '
>>> s += 'and the number of the counting shall be three'
>>> b = s.encode()                        # convert to utf-8
>>> len(b)
89
>>> c = gzip.compress(b)
>>> len(c)
77
>>> gzip.decompress(c).decode()[:42]      # decompress and convert to
text
'Three shall be the number thou shalt count'
```

Кроме того, класс `zipfile.ZipExtFile` был переработан внутренне для представления файлов, хранящихся внутри архива. Новая реализация значительно быстрее и может быть обернута в объект `io.BufferedReader` для большего ускорения. Это также решает проблему, из-за которой чередующиеся вызовы `read` и `readline` давали неправильные результаты.

tarfile

Класс `TarFile` теперь можно использовать в качестве контекстного менеджера. Кроме того, его метод `add()` имеет новую опцию `filter`, которая управляет тем, какие файлы добавляются в архив, и позволяет редактировать метаданные файла.

Новый параметр `filter` заменяет старый, менее гибкий параметр `exclude`, который теперь устарел. Если указано, необязательный параметр `filter` должен быть аргументом ключевого слова.

Предоставляемая пользователем функция фильтра принимает объект `TarInfo` и возвращает обновленный объект `TarInfo`, или, если он хочет, чтобы файл был исключен, функция может вернуть `None`:

```
>>> import tarfile, glob

>>> def myfilter(tarinfo):
...     if tarinfo.isfile():             # only save real files
...         tarinfo.uname = 'monty'      # redact the user name
...         return tarinfo

>>> with tarfile.open(name='myarchive.tar.gz', mode='w:gz') as tf:
...     for filename in glob.glob('*.txt'):
...         tf.add(filename, filter=myfilter)
...     tf.list()
-rw-r--r-- monty/501        902 2011-01-26 17:59:11 annotations.txt
-rw-r--r-- monty/501        123 2011-01-26 17:59:11
general_questions.txt
-rw-r--r-- monty/501       3514 2011-01-26 17:59:11 prion.txt
-rw-r--r-- monty/501        124 2011-01-26 17:59:11 py_todo.txt
-rw-r--r-- monty/501       1399 2011-01-26 17:59:11
semaphore_notes.txt
```

hashlib

Модуль `hashlib` имеет два новых постоянных атрибута, в которых перечислены алгоритмы хеширования, которые гарантированно присутствуют во всех реализациях, и те, которые доступны в текущей реализации:

```
>>> import hashlib

>>> hashlib.algorithms_guaranteed
{'sha1', 'sha224', 'sha384', 'sha256', 'sha512', 'md5'}

>>> hashlib.algorithms_available
{'md2', 'SHA256', 'SHA512', 'dsaWithSHA', 'mdc2', 'SHA224', 'MD4',
'sha256',
'sha512', 'ripemd160', 'SHA1', 'MDC2', 'SHA', 'SHA384', 'MD2',
'ecdsa-with-SHA1','md4', 'md5', 'sha1', 'DSA-SHA', 'sha224',
'dsaEncryption', 'DSA', 'RIPEMD160', 'sha', 'MD5', 'sha384'}
```

ast

Модуль `ast` обладает замечательным инструментом общего назначения для безопасного вычисления строковых выражений с использованием синтаксиса литералов Python. Функция `last.literal_eval()` служит безопасной альтернативой встроенной функции `eval()`, которой легко злоупотреблять. Python 3.2 добавляет литералы `bytes` и `set` в список поддерживаемых типов: строки, байты, числа, кортежи, списки, словари, множества, логические значения и `None`.

```
>>> from ast import literal_eval

>>> request = "{'req': 3, 'func': 'pow', 'args': (2, 0.5)}"
>>> literal_eval(request)
{'args': (2, 0.5), 'req': 3, 'func': 'pow'}

>>> request = "os.system('do something harmful')"
>>> literal_eval(request)
Traceback (most recent call last):
  ...
ValueError: malformed node or string: <_ast.Call object at
0x101739a10>
```

os

Различные операционные системы используют различные кодировки для имен файлов и переменных среды. Модуль `os` предоставляет две новые функции, `fsencode()` и `fsdecode()`, для кодирования и декодирования имен файлов:

```
>>> import os
>>> filename = 'Sehenswürdigkeiten'
>>> os.fsencode(filename)
b'Sehensw\xc3\xbcrdigkeiten'
```

Некоторые операционные системы допускают прямой доступ к закодированным байтам в среде. Если это так, константа `os.supports_bytes_environ` будет иметь значение `true`.

Для прямого доступа к закодированным переменным среды (если таковые имеются) используйте новую функцию `os.getenv()` или используйте `os.environb`, которая является байтовой версией `os.environ`.

shutil

Функция `shutil.copytree()` имеет две новые опции:

- `ignore_dangling_symlinks`: когда `symlinks=False`, чтобы функция копировала файл, на который указывает символическая ссылка, а не саму символическую ссылку. Этот параметр отключит сообщение об ошибке, если файл не существует.
- `copy_function`: вызываемая функция, которая будет использоваться для копирования файлов. `shutil.copy2()` используется по умолчанию.

Кроме того, модуль `shutil` теперь поддерживает операции архивирования zip-файлов, несжатых и сжатых (gzipped, bzipped) файлов `tar`. И есть функции для регистрации дополнительных форматов архивных файлов (таких как xz сжатые tar-файлы или пользовательские форматы). Основными функциями являются `make_archive()` и `unpack_archive()`. По умолчанию обе работают с текущим каталогом (который может быть установлен с помощью `os.chdir()`) и с любыми подкаталогами. Имя файла архива должно быть указано с полным путем. Этап архивирования является неразрушающим (исходные файлы остаются неизменными).

```
>>> import shutil, pprint

>>> os.chdir('mydata')  # change to the source directory
>>> f = shutil.make_archive('/var/backup/mydata',
...                         'zip')      # archive the current
directory
>>> f                                   # show the name of archive
'/var/backup/mydata.zip'
>>> os.chdir('tmp')                     # change to an unpacking
>>> shutil.unpack_archive('/var/backup/mydata.zip')  # recover the
data

>>> pprint.pprint(shutil.get_archive_formats())  # display known
formats
[('bztar', "bzip2'ed tar-file"),
 ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'),
 ('zip', 'ZIP file')]

>>> shutil.register_archive_format(     # register a new archive
format
...     name='xz',
...     function=xz.compress,           # callable archiving function
...     extra_args=[('level', 8)],      # arguments to the function
...     description='xz compression'
... )
```

sqlite3

Модуль `sqlite3` был обновлен до версии 2.6.0 pysqlite. У него есть две новые возможности.

- Атрибут `sqlite3.Connection.in_transit` имеет значение `true`, если существует активная транзакция для незафиксированных изменений.
- Методы `sqlite3.Connection.enable_load_extension()` и `sqlite3.Connection.load_extension()` позволяют загружать расширения SQLite из файлов ".so". Одним из хорошо известных расширений является расширение полнотекстового поиска, распространяемое вместе с SQLite.

html

Был представлен новый модуль `html`, содержащий только одну функцию `escape()`, которая используется для экранирования зарезервированных символов из HTML-разметки:

```
>>> import html
>>> html.escape('x > 2 && x < 7')
'x &gt; 2 &amp;&amp; x &lt; 7'
```

socket

Модуль `socket` имеет два новых улучшения:

- Объекты сокетов теперь имеют метод `detach()`, который переводит сокет в закрытое состояние без фактического закрытия базового файлового дескриптора. Последний затем может быть повторно использован для других целей.
- `socket.create_connection()` теперь поддерживает протокол управления контекстом для безоговорочного использования исключений `socket.error` и закрытия сокета по завершении.

ssl

Модуль `ssl` добавил ряд функций для удовлетворения общих требований к безопасным (зашифрованным, прошедшим проверку подлинности) интернет-соединениям:

- Новый класс, `SSLContext`, служит контейнером для постоянных данных SSL, таких как настройки протокола, сертификаты, закрытые ключи и различные другие параметры. Он включает в себя `wrap_socket()` для создания сокета SSL из контекста SSL.
- Новая функция, `ssl.match_hostname()`, поддерживает проверку подлинности сервера для протоколов более высокого уровня, реализуя правила HTTPS (из RFC 2818), которые также подходят для других протоколов.
- Функция конструктора `ssl.wrap_socket()` теперь принимает аргумент `ciphers`. В строке `ciphers` перечислены разрешенные алгоритмы шифрования, использующие формат, описанный в документации OpenSSL.
- При подключении к последним версиям OpenSSL модуль `ssl` теперь поддерживает расширение указания имени сервера для протокола TLS, позволяя нескольким “виртуальным хостам” использовать разные сертификаты на одном IP-порту. Это расширение поддерживается только в клиентском режиме и активируется путем передачи аргумента `server_hostname` в `ssl.SSLContext.wrap_socket()`.
- В модуль `ssl` были добавлены различные опции, такие как `OP_NO_SSLv2`, которая отключает небезопасный и устаревший протокол SSLv2.
- Расширение теперь загружает все шифры OpenSSL и алгоритмы дайджеста. Если некоторые SSL-сертификаты не могут быть проверены, о них сообщается как об ошибке “неизвестный алгоритм”.
- Используемая версия OpenSSL теперь доступна с помощью атрибутов модуля `ssl.OPENSSL_VERSION` (строка), `ssl.OPENSSL_VERSION_INFO` (5-кратный кортеж) и `ssl.OPENSSL_VERSION_NUMBER` (целое число).

nntp

Модуль `nntplib` имеет обновленную реализацию с улучшенной семантикой байтов и текста, а также более практичными API. Эти улучшения нарушают совместимость с версией `nntplib` в Python 3.1, которая сама по себе была частично неработоспособной.

Поддержка безопасных подключений как неявных (с использованием `nntplib.NNTP_SSL`), так и явных (с использованием `nntplib.NNTP.starttls()`). Также был добавлен протокол TLS.

certificates

`http.клиент.HTTPSConnection`, `urllib.request.HTTPSHandler` и `urllib.request.urlopen()` теперь принимают необязательные аргументы, позволяющие проверять сертификат сервера на соответствие набору центров сертификации, как рекомендуется при общедоступном использовании HTTPS.

imaplib

Поддержка явного протокола TLS для стандартных подключений IMAP4 была добавлена с помощью нового метода `imaplib.IMAP4.starttls`.

http.client

В модуле `http.client` был внесен ряд небольших улучшений API. Простые ответы HTTP 0.9 старого стиля больше не поддерживаются, а параметр `strict` устарел во всех классах.

Классы `HttpConnection` и `HTTPSConnection` теперь имеют параметр `source_address` для кортежа (хост, порт), указывающий, откуда осуществляется HTTP-соединение.

В HTTPS-соединение была добавлена поддержка проверки сертификатов и виртуальных хостов HTTPS.

Метод `request()` для объектов подключения допускал необязательный аргумент `body`, чтобы для предоставления содержимого запроса можно было использовать объект `file`. Удобно, что аргумент `body` теперь также принимает итерируемый объект, если он включает явный заголовок `Content-Length`. Этот расширенный интерфейс гораздо более гибкий, чем раньше.

To establish an HTTPS connection through a proxy server, there is a new set_tunnel() method that sets the host and port for HTTP Connect tunneling. Чтобы установить HTTPS-соединение через прокси-сервер, существует новый метод `set_tunnel()`, который задает хост и порт для туннелирования соединения HTTP.

Чтобы соответствовать поведению `http.server`, клиентская библиотека HTTP теперь также кодирует заголовки в кодировке ISO-8859-1 (Latin-1). Она уже делала это для входящих заголовков, так что теперь поведение согласовано как для входящего, так и для исходящего трафика.

unittest

Модуль `unittest` имеет ряд улучшений, поддерживающих обнаружение тестов для пакетов, упрощение экспериментов в интерактивной подсказке, новые методы `testcase`, улучшенные диагностические сообщения о сбоях тестирования и улучшенные имена методов.

- Вызов командной строки `python -m unittest` теперь может принимать пути к файлам вместо имен модулей для выполнения определенных тестов (bpo-10620). Новое обнаружение тестов может находить тесты внутри пакетов, определяя местоположение любого теста, который можно импортировать из каталога верхнего уровня. Каталог верхнего уровня можно указать с помощью параметра `-t`, шаблон для сопоставления файлов с `-p` и каталог для начала обнаружения с `-s`:

```
$ python -m unittest discover -s my_proj_dir -p _test.py
```

- Экспериментировать с интерактивной подсказкой теперь проще, потому что класс `unittest.case.TestCase` теперь может быть создан без аргументов:

```
>>> from unittest import TestCase
>>> TestCase().assertEqual(pow(2, 3), 8)
```

- Модуль `unittest` имеет два новых метода, `assertWarns()` и `assertWarnsRegex()` для проверки того, что данный тип предупреждения запускается тестируемым кодом:

```
with self.assertWarns(DeprecationWarning):
    legacy_function('XYZ')
```

Другой новый метод, `assertCountEqual()` используется для сравнения двух итераций, чтобы определить, равны ли их количества элементов (присутствуют ли одни и те же элементы с одинаковым количеством вхождений независимо от порядка):

```
def test_anagram(self):
    self.assertCountEqual('algorithm', 'logarithm')
```

Основной особенностью модуля `unittest` является попытка произвести значимую диагностику в случае сбоя теста. Когда это возможно, сбой записывается вместе с различием выходных данных. Это особенно полезно для анализа файлов журналов неудачных тестовых запусков. Однако, поскольку различия иногда могут быть объемными, появился новый атрибут `maxDiff`, который устанавливает максимальную длину отображаемых различий.

Кроме того, имена методов в модуле подверглись ряду очисток.

Например, `assertregexp()` - это новое имя для `assertRegexpMatches()`, которое было неправильно названо, потому что тест использует `re.search()`, а не `re.match()`. Другие методы, использующие регулярные выражения, теперь называются с использованием краткой формы “Regex” вместо “Regexp” – это соответствует именам, используемым в других реализациях `unittest`, соответствует старому имени Python для модуля `re` и имеет однозначную `camel`-оболочку.

Чтобы улучшить согласованность, некоторые давние псевдонимы методов устарели в пользу предпочтительных имен:

Старое имя	**Предпочтительное имя**
`assert_()`	`assertTrue()`
`assertEquals()`	`assertEqual()`
`assertNotEquals()`	`assertNotEqual()`
`assertAlmostEquals()`	`assertAlmostEqual()`
`assertNotAlmostEquals()`	`assertNotAlmostEqual()`

Аналогично, ожидается, что методы `TestCase.fail*`, устаревшие в Python 3.1, будут удалены в Python 3.3. Также смотрите раздел "Устаревшие псевдонимы" в документации `unittest`.

Метод `assertDictContainsSubset()` был признан устаревшим, поскольку он был неправильно реализован с аргументами в неправильном порядке. Это создавало трудноотлаживаемые оптические иллюзии, когда тесты, подобные `TestCase().assertDictContainsSubset({'a':1, 'b':2}, {'a':1})`, завершались неудачей.

random

Целочисленные методы в модуле `random` теперь лучше справляются с получением равномерных распределений. Ранее они вычисляли выборки с помощью `int (n * random())`, которые имели небольшое смещение всякий раз, когда `n` не было степенью двойки. Теперь производится множественный выбор из диапазона до следующей степени двойки, и выбор сохраняется только тогда, когда он попадает в диапазон `0 <= x < n`. Затронутыми функциями и методами являются `randrange()`, `randint()`, `choice()`, `shuffle()` и `sample()`.

poplib

Класс `POP3_SSL` теперь принимает параметр `context`, который представляет собой объект `ssl.SSLContext`, позволяющий объединять параметры конфигурации SSL, сертификаты и закрытые ключи в единую (потенциально долговечную) структуру.

asyncore

`asyncore.dispatcher` теперь предоставляет метод `handle_accepted()`, возвращающий пару (`sock`, `addr`), которая вызывается, когда соединение фактически установлено с новой удаленной конечной точкой. Предполагается, что это используется в качестве замены старого `handle_accept()` и позволяет избежать прямого вызова `accept()` пользователем.

tempfile

Модуль `tempfile` имеет новый контекстный менеджер, временный каталог, который обеспечивает легкую детерминированную очистку временных каталогов:

```
with tempfile.TemporaryDirectory() as tmpdirname:
    print('created temporary dir:', tmpdirname)
```

inspect

Модуль `inspect` имеет новую функцию `getgeneratorstate()`, позволяющую легко идентифицировать текущее состояние генератора-итератора:

```
>>> from inspect import getgeneratorstate
>>> def gen():
...     yield 'demo'
>>> g = gen()
>>> getgeneratorstate(g)
'GEN_CREATED'
>>> next(g)
'demo'
>>> getgeneratorstate(g)
'GEN_SUSPENDED'
>>> next(g, None)
>>> getgeneratorstate(g)
'GEN_CLOSED'
```

Для поддержки поиска без возможности активации динамического атрибута в модуле `inspect` появилась новая функция `getattr_static()`. В отличие от `hasattr()`, это настоящий поиск только для чтения, гарантированно не изменяющий состояние во время поиска:

```
>>> class A:
...     @property
...     def f(self):
...         print('Running')
...         return 10
...
>>> a = A()
>>> getattr(a, 'f')
Running
10
>>> inspect.getattr_static(a, 'f')
<property object at 0x1022bd788>
```

pydoc

Модуль `pydoc` теперь предоставляет значительно улучшенный интерфейс веб-сервера, а также новую опцию командной строки `-b` для автоматического открытия окна браузера для отображения этого сервера:

```
$ pydoc3.2 -b
```

dis

Модуль `dis` получил две новые функции для проверки кода: `code_info()` и `show_code()`. Обе предоставляют подробную информацию об объекте кода для предоставленной функции, метода, строки исходного кода или объекта кода. Первый возвращает строку, а второй печатает ее:

```
>>> import dis, random
>>> dis.show_code(random.choice)
Name:              choice
Filename:
/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/rando
m.py
Argument count:    2
Kw-only arguments: 0
Number of locals:  3
Stack size:        11
Flags:             OPTIMIZED, NEWLOCALS, NOFREE
Constants:
```

```
   0: 'Choose a random element from a non-empty sequence.'
   1: 'Cannot choose from an empty sequence'
Names:
   0: _randbelow
   1: len
   2: ValueError
   3: IndexError
Variable names:
   0: self
   1: seq
   2: i
```

Кроме того, функция `dis()` теперь принимает строковые аргументы, так что обычная идиома `dis(compile(s, '', 'eval'))` может быть сокращена до `dis(s)`:

```
>>> dis('3*x+1 if x%2==1 else x//2')
  1           0 LOAD_NAME                0 (x)
              3 LOAD_CONST               0 (2)
              6 BINARY_MODULO
              7 LOAD_CONST               1 (1)
             10 COMPARE_OP               2 (==)
             13 POP_JUMP_IF_FALSE       28
             16 LOAD_CONST               2 (3)
             19 LOAD_NAME                0 (x)
             22 BINARY_MULTIPLY
             23 LOAD_CONST               1 (1)
             26 BINARY_ADD
             27 RETURN_VALUE
        >>   28 LOAD_NAME                0 (x)
             31 LOAD_CONST               0 (2)
             34 BINARY_FLOOR_DIVIDE
             35 RETURN_VALUE
```

Взятые вместе, эти улучшения облегчают изучение того, как реализован CPython, и позволяют самим увидеть, что делает синтаксис языка “под капотом”.

dbm

Все модули базы данных теперь поддерживают методы `get()` и `setdefault()`.

ctypes

Новый тип, `ctypes.c_size_t`, представляет тип данных C `ssize_t`.

site

Модуль сайта имеет три новые функции, полезные для представления отчетов о деталях данной установки Python.

- `getsitepackages()` перечисляет все глобальные каталоги пакетов сайта.
- `getuserbase()` сообщает о базовом каталоге пользователя, в котором могут храниться данные.
- `getusersitepackages()` показывает путь к каталогу пакетов сайта для конкретного пользователя.

```
>>> import site
>>> site.getsitepackages()
['/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/sit
e-packages',
 '/Library/Frameworks/Python.framework/Versions/3.2/lib/site-python',
 '/Library/Python/3.2/site-packages']
>>> site.getuserbase()
'/Users/raymondhettinger/Library/Python/3.2'
>>> site.getusersitepackages()
'/Users/raymondhettinger/Library/Python/3.2/lib/python/site-packages'
```

Удобно, что некоторые функции сайта доступны непосредственно из командной строки:

```
$ python -m site --user-base
/Users/raymondhettinger/.local
$ python -m site --user-site
/Users/raymondhettinger/.local/lib/python3.2/site-packages
```

sysconfig

Новый модуль `sysconfig` упрощает поиск путей установки и переменных конфигурации, которые различаются в зависимости от платформ и установок.

Модуль предлагает доступ к простым функциям доступа к информации о платформе и версии:

- `get_platform()` возвращает значения, такие как linux-i586 или macosx-10.6-ppc.
- `get_python_version()` возвращает строку версии Python, такую как “3.2”.

Он также предоставляет доступ к путям и переменным, соответствующим одной из семи именованных схем, используемых `distutils`. К ним относятся

`posix_prefix`, `posix_home`, `posix_user`, `nt`, `nt_user`, `os2`, `os2_home`:

- `get_paths()` создает словарь, содержащий пути установки для текущей схемы установки.
- `get_config_vars()` возвращает словарь переменных, специфичных для конкретной платформы.

Существует также удобный интерфейс командной строки:

```
C:\Python32>python -m sysconfig
Platform: "win32"
Python version: "3.2"
Current installation scheme: "nt"

Paths:
        data = "C:\Python32"
        include = "C:\Python32\Include"
        platinclude = "C:\Python32\Include"
        platlib = "C:\Python32\Lib\site-packages"
        platstdlib = "C:\Python32\Lib"
        purelib = "C:\Python32\Lib\site-packages"
        scripts = "C:\Python32\Scripts"
        stdlib = "C:\Python32\Lib"

Variables:
        BINDIR = "C:\Python32"
        BINLIBDEST = "C:\Python32\Lib"
        EXE = ".exe"
        INCLUDEPY = "C:\Python32\Include"
        LIBDEST = "C:\Python32\Lib"
        SO = ".pyd"
        VERSION = "32"
        abiflags = ""
        base = "C:\Python32"
        exec_prefix = "C:\Python32"
        platbase = "C:\Python32"
        prefix = "C:\Python32"
        projectbase = "C:\Python32"
        py_version = "3.2"
        py_version_nodot = "32"
        py_version_short = "3.2"
        srcdir = "C:\Python32"
        userbase = "C:\Documents and Settings\Raymond\Application
Data\Python"
```

pdb

Модуль `pdb` получил ряд улучшений в удобстве использования:

- `pdb.py` теперь есть опция а `-c`, которая выполняет команды, указанные в файле сценария `.pdbrc`.
- Файл сценария `.pdbrc` может содержать команды `continue` и `next`, которые продолжают отладку.
- Конструктор класса `Pdb` теперь принимает аргумент `nosigint`.
- Новые команды: `l(list)`, `ll(long list)` и `source` для перечисления исходного кода.
- Новые команды: `display` и `undisplay` для отображения или скрытия значения выражения, если оно изменилось.
- Новая команда: `interact` для запуска интерактивного интерпретатора, содержащего глобальные и локальные имена, найденные в текущей области.
- Точки останова могут быть очищены по номеру точки останова.

configparser

Модуль `configparser` был изменен для улучшения удобства использования и предсказуемости анализатора по умолчанию и его поддерживаемого синтаксиса INI. Старый класс `ConfigParser` был удален в пользу `SafeConfigParser`, который, в свою очередь, был переименован в `ConfigParser`. Поддержка встроенных комментариев теперь отключена по умолчанию, и дублирование разделов или опций не допускается в одном источнике конфигурации.

Анализаторы конфигурации получили новый API, основанный на протоколе сопоставления:

```
>>> parser = ConfigParser()
>>> parser.read_string("""
... [DEFAULT]
... location = upper left
... visible = yes
... editable = no
... color = blue
...
... [main]
... title = Main Menu
... color = green
...
... [options]
... title = Options
... """)
```

```
>>> parser['main']['color']
'green'
>>> parser['main']['editable']
'no'
>>> section = parser['options']
>>> section['title']
'Options'
>>> section['title'] = 'Options (editable: %(editable)s)'
>>> section['title']
'Options (editable: no)'
```

Новый API реализован поверх классического API, поэтому подклассы пользовательских анализаторов должны иметь возможность использовать его без изменений.

Структура файла INI, принятая анализаторами конфигурации, теперь может быть настроена. Пользователи могут указать альтернативные разделители параметров/значений и префиксы комментариев, изменить название раздела по умолчанию или переключить синтаксис интерполяции.

Существует поддержка подключаемой интерполяции включая дополнительный обработчик интерполяции `ExtendedInterpolation`:

```
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> parser.read_dict({'buildout': {'directory': '/home/ambv/zope9'},
...                   'custom': {'prefix': '/usr/local'}})
>>> parser.read_string("""
... [buildout]
... parts =
...   zope9
...   instance
... find-links =
...   ${buildout:directory}/downloads/dist
...
... [zope9]
... recipe = plone.recipe.zope9install
... location = /opt/zope
...
... [instance]
... recipe = plone.recipe.zope9instance
... zope9-location = ${zope9:location}
... zope-conf = ${custom:prefix}/etc/zope.conf
... """)
>>> parser['buildout']['find-links']
'\n/home/ambv/zope9/downloads/dist'
>>> parser['instance']['zope-conf']
'/usr/local/etc/zope.conf'
```

```
>>> instance = parser['instance']
>>> instance['zope-conf']
'/usr/local/etc/zope.conf'
>>> instance['zope9-location']
'/opt/zope'
```

Также был введен ряд более мелких функций, таких как поддержка указания кодировки в операциях чтения, указания резервных значений для get-функций или чтения непосредственно из словарей и строк.

urllib.parse

Для модуля `urllib.parse` был внесен ряд улучшений для удобства использования.

Функция `urlparse()` теперь поддерживает адреса IPv6, как описано в RFC 2732:

```
>>> import urllib.parse
>>>
urllib.parse.urlparse('http://[dead:beef:cafe:5417:affe:8FA3:deaf:fee
d]/foo/')
ParseResult(scheme='http',
            netloc='[dead:beef:cafe:5417:affe:8FA3:deaf:feed]',
            path='/foo/',
            params='',
            query='',
            fragment='')
```

Функция `urldefrag()` теперь возвращает именованный кортеж:

```
>>> r = urllib.parse.urldefrag('http://python.org/about/#target')
>>> r
DefragResult(url='http://python.org/about/', fragment='target')
>>> r[0]
'http://python.org/about/'
>>> r.fragment
'target'
```

Кроме того, функция `urlencode()` теперь стала гораздо более гибкой, принимая в качестве аргумента запроса либо строку, либо байты. Если это строка, то параметры `safe`, `encoding` и `error` отправляются в `quote_plus()` для кодирования:

```
>>> urllib.parse.urlencode([
...      ('type', 'telenovela'),
...      ('name', '¿Dónde Está Elisa?')],
...      encoding='latin-1')
'type=telenovela&name=%BFD%F3nde+Est%E1+Elisa%3F'
```

Как подробно описано в разделе Синтаксический анализ байтов в кодировке ASCII, все функции `urllib.parse` теперь принимают строки байтов в кодировке ASCII в качестве входных данных, если они не смешаны с обычными строками. Если в качестве параметров заданы байтовые строки в кодировке ASCII, возвращаемые типы также будут байтовыми строками в кодировке ASCII:

```
>>> urllib.parse.urlparse(b'http://www.python.org:80/about/')
ParseResultBytes(scheme=b'http', netloc=b'www.python.org:80',
                 path=b'/about/', params=b'', query=b'',
                 fragment=b'')
```

mailbox

Благодаря совместным усилиям Р. Дэвида Мюррея модуль почтового ящика был исправлен для Python 3.2. Проблема заключалась в том, что почтовый ящик изначально был разработан с текстовым интерфейсом, но сообщения электронной почты лучше всего представлять байтами, поскольку различные части сообщения могут иметь разные кодировки.

Решение использовало двоичную поддержку пакета электронной почты для анализа произвольных сообщений электронной почты. Кроме того, решение потребовало ряда изменений в API.

Как и ожидалось, метод `add()` для объектов `mailbox.Mailbox` теперь принимает двоичный ввод.

`StringIO` и ввод текстового файла устарел. Кроме того, ввод строки завершится преждевременным сбоем, если используются символы, отличные от ASCII. Ранее это приводило к сбою, когда электронное письмо обрабатывалось на более позднем этапе.

Существует также поддержка двоичного вывода. Метод `get_file()` теперь возвращает файл в двоичном режиме (где раньше он неправильно переводил файл в текстовый режим). Существует также новый метод `get_bytes()`, который возвращает байтовое представление сообщения, соответствующего данному ключу.

По-прежнему возможно получить недвоичный вывод, используя метод `get_string()` старого API, но такой подход не очень полезен. Вместо этого лучше всего извлекать сообщения из объекта `Message` или загружать их из двоичного ввода.

turtledemo

Демонстрационный код для модуля `turtle` был перенесен из демонстрационного каталога в основную библиотеку. Он включает в себя более десятка примеров сценариев с живыми дисплеями. Находясь на `sys.path`, теперь его можно запустить непосредственно из командной строки:

```
$ python -m turtledemo
```

Многопоточность

Механизм сериализации выполнения одновременно запущенных потоков Python (обычно известный как GIL или Глобальная блокировка интерпретатора) был переписан. Среди целей были более предсказуемые интервалы переключения и снижение накладных расходов из-за конфликта блокировок и количества последующих системных вызовов. Понятие “контрольного интервала”, позволяющего переключать потоки, было оставлено и заменено абсолютной продолжительностью, выраженной в секундах. Этот параметр настраивается с помощью `sys.setswitchinterval()`. В настоящее время значение по умолчанию равно 5 миллисекундам.

Дополнительные сведения о реализации можно прочитать из сообщения списка рассылки python-dev (однако “приоритетные запросы”, представленные в этом сообщении, не были сохранены для включения).

Регулярные и рекурсивные блокировки теперь принимают необязательный аргумент `timeout` для своего метода `acquire()`.

Similarly, threading.Semaphore.acquire() also gained a timeout argument. (Contributed by Torsten Landschoff; bpo-850728.)
Аналогично, `threading.Semaphore.acquire()` также получил аргумент `timeout`.

Регулярное и рекурсивное получение блокировок теперь может быть прервано сигналами на платформах, использующих `Pthreads`. Это означает, что программы на Python, которые заходят в тупик при получении блокировок, могут быть успешно завершены путем многократной отправки SIGINT процессу (нажатием Ctrl + C в большинстве оболочек).

Оптимизации

Был добавлен ряд небольших улучшений производительности:

- Оптимизатор peephole в Python теперь распознает шаблоны, такие как `x in {1, 2, 3}`, как тест на принадлежность к набору констант. Оптимизатор преобразует множество в `frozenset` и сохраняет предварительно созданную константу. Теперь, когда ограничение по быстродействию исчезло, практично начать писать тесты членов, используя set-нотацию. Этот стиль одновременно семантически понятен и быстр в эксплуатации:

  ```
  extension = name.rpartition('.')[2]
  if extension in {'xml', 'html', 'xhtml', 'css'}:
      handle(name)
  ```

- Сериализация и десериализация данных с помощью модуля `pickle` теперь выполняется в несколько раз быстрее.

- Алгоритм `Timsort`, используемый в `list.sort()` и `sorted()`, теперь работает быстрее и использует меньше памяти при вызове с помощью ключевой функции. Ранее каждый элемент списка был обернут временным объектом, который запоминал значение ключа, связанное с каждым элементом. Теперь два массива ключей и значений сортируются параллельно. Это экономит память, потребляемую оболочками сортировки, и экономит время, потерянное на делегирование сравнений.

- Производительность декодирования JSON повышается, а потребление памяти уменьшается всякий раз, когда одна и та же строка повторяется для нескольких ключей. Кроме того, кодировка JSON теперь использует ускорения C, когда аргумент `sort_keys` имеет значение `true`.

- Рекурсивные блокировки (созданные с помощью `threading.RLock()` `API`) теперь выигрывают от реализации на C, которая делает их такими же быстрыми, как обычные блокировки, и в 10-15 раз быстрее, чем их предыдущая реализация на чистом Python.

- Алгоритм быстрого поиска в `stringlib` теперь используется методами `split()`, `rsplit()`, `splitlines()` и `replace()` для объектов `bytes`, `bytearray` и `str`. Аналогично, алгоритм также используется `rfind()`, `rindex()`, `rsplit()` и `rpartition()`.

- Преобразования целого числа в строку теперь работают с двумя “цифрами” одновременно, уменьшая количество операций деления и по модулю.

Было сделано еще несколько незначительных оптимизаций. Разность множеств теперь выполняется быстрее, когда один операнд намного больше другого (исправление Андресса Беннеттса в bpo-8685). Метод `array.repeat()` имеет более быструю реализацию (bpo-1569291 Александра Белопольского). `BaseHTTPRequestHandler` имеет более эффективную буферизацию (bpo-3709 от Эндрю Шаафа). Функция `operator.attrgetter()` была ускорена (bpo-10160 от Christos Georgiou). И `ConfigParser` загружает многострочные аргументы немного быстрее (bpo-7113 от Лукаша Ланги).

Unicode

Python был обновлен до Unicode 6.0.0. Обновление стандарта добавляет более 2000 новых символов, включая эмодзи, которые важны для мобильных телефонов.

Кроме того, обновленный стандарт изменил свойства символов для двух символов Kannada (U+0CF1, U+0CF2) и одного нового цифрового символа New Tai Lue (U+19DA), что делает первый допустимым для использования в идентификаторах, в то время как второй дисквалифицирует. Дополнительные сведения см. в разделе Изменения базы данных символов Юникода.

Кодировки

Добавлена поддержка арабской кодировки DOS cp720 (bpo-1616979).

Кодировка MBCS больше не игнорирует аргумент обработчика ошибок. В строгом режиме по умолчанию он выдает ошибку `UnicodeDecodeError` при обнаружении некодируемой последовательности байтов и ошибку `UnicodeEncodeError` для некодируемого символа.

Кодек MBCS поддерживает обработчики ошибок `'strict'` и `'ignore'` для декодирования, а также `'strict'` и `'replace'` для кодирования.

Чтобы эмулировать кодировку Python3.1 MBCS, выберите обработчик `'ignore'` для декодирования и обработчик `'replace'` для кодирования.

В Mac OS X Python декодирует аргументы командной строки с помощью `'utf-8'`, а не кодировки локали.

По умолчанию `tarfile` использует кодировку `'utf-8'` в Windows (вместо `'mbcs'`) и обработчик ошибок `'surrogateescape'` во всех операционных системах.

Документация

Документация продолжает совершенствоваться.

- Таблица быстрых ссылок была добавлена в начало длинных разделов, таких как встроенные функции. В случае `itertools` ссылки сопровождаются таблицами сводок в стиле `cheatsheet`, чтобы обеспечить обзор и освежить память без необходимости читать все документы.

- В некоторых случаях чистый исходный код Python может быть полезным дополнением к документации, поэтому теперь многие модули содержат быстрые ссылки на последнюю версию исходного кода. Например, документация по модулю `functools` содержит быструю ссылку вверху с надписью:

  `Source code Lib/functools.py`

- Документы теперь содержат больше примеров и рецептов. В частности, модуль `re` имеет обширный раздел "Примеры регулярных выражений". Аналогичным образом, модуль `itertools` продолжает обновляться новыми рецептами `Itertools`.

- Модуль `datetime` теперь имеет вспомогательную реализацию на чистом Python. Никакая функциональность не была изменена. Это просто обеспечивает более легкую для чтения альтернативную реализацию.

- Неподдерживаемый каталог `Demo` был удален. Некоторые демо-версии были интегрированы в документацию, некоторые были перемещены в каталог `Tools/demo`, а другие были полностью удалены.

IDLE

- В меню формат теперь есть опция очистки исходных файлов путем удаления завершающих пробелов.
- IDLE в Mac OS X теперь работает как с Carbon AquaTk, так и с Cocoa AquaTk.

Code Repository

В дополнение к существующему репозиторию кода Subversion на http://svn.python.org в настоящее время существует репозиторий Mercurial по адресу https://hg.python.org/.

После выпуска версии 3.2 планируется перейти на Mercurial в качестве основного репозитория. Эта распределенная система контроля версий должна облегчить членам сообщества создание внешних наборов изменений и обмен ими. Подробности см. в PEP 385.

Чтобы научиться использовать новую систему контроля версий, ознакомьтесь с Кратким началом или руководством по рабочим процессам Mercurial.

Изменения в сборке и C API

Изменения в процессе сборки Python и в C API включают:

- Сценарии `idle`, `pydoc` и `2to3` теперь устанавливаются с суффиксом, зависящим от версии, на `make altinstall` (bpo-10679).
- Функции C, которые обращаются к базе данных Unicode, теперь принимают и возвращают символы из полного диапазона Unicode, даже в узких сборках unicode Py_UNICODE_TOLOWER, Py_UNICODE_ISDECIMAL и другие). Видимое отличие в Python заключается в том, что `unicodedata.numeric()` теперь возвращает правильное значение для больших кодовых точек, а `repr()` может рассматривать большее количество символов как доступные для печати.
- Вычисляемые `goto` теперь включены по умолчанию в поддерживаемых компиляторах (которые определяются сценарием настройки). Они все еще могут быть отключены выборочно, указав `--without-computed-goto`.
- Опция `--with-wctype-functions` была удалена. Встроенная база данных unicode теперь используется для всех функций.
- Хэш-значения теперь являются значениями нового типа, `Py_hash_t`, который определяется как тот же размер, что и указатель. Ранее они имели тип `long`, который в некоторых 64-разрядных операционных системах по-прежнему имеет длину всего 32 бита. В результате этого исправления `set` и `dict` теперь могут содержать более 2**32 записей в сборках с 64-разрядными указателями (ранее они могли увеличиваться до такого размера, но их производительность катастрофически снижалась).
- Новый макрос `Py_VA_COPY` копирует состояние списка аргументов переменной. Он эквивалентен C99 `va_copy`, но доступен на всех платформах Python (bpo-2443).
- Новая функция C API `PySys_SetArgvEx()` позволяет встроенному интерпретатору устанавливать `sys.argv` без изменения `sys.path` (bpo-5753).
- `PyEval_CallObject` теперь доступен только в форме макроса. Объявление функции, которое было сохранено по соображениям обратной совместимости, теперь удалено – макрос был введен в 1997 году (bpo-8276).

- Появилась новая функция `PyLong_AsLongLongAndOverflow()`, которая аналогична `PyLong_AsLongAndOverflow()`. Они обе служат для преобразования Python int в собственный тип с фиксированной шириной, обеспечивая при этом обнаружение случаев, когда преобразование не подходит (bpo-7767).
- Функция `PyUnicode_CompareWithASCIIString()` теперь возвращает значение `not equal`, если строка Python завершается нулем.
- Существует новая функция `PyErr_NewExceptionWithDoc()`, которая похожа на `PyErr_NewException()`, но позволяет указывать строку документа. Это позволяет исключениям C иметь те же возможности самодокументирования, что и их аналоги на чистом Python (bpo-7033).
- При компиляции с параметром `--with-valgrind` распределитель `pymalloc` будет автоматически отключен при запуске под управлением Valgrind. Это обеспечивает улучшенное обнаружение утечек памяти при запуске под управлением Valgrind, в то же время используя преимущества `pymalloc` в другое время (bpo-2422).
- Удалён формат `O?` из функций `PyArg_Parse`. Формат больше не используется, и он никогда не был задокументирован (bpo-8837).

В C-API был внесен ряд других небольших изменений. Смотрите файл Misc/NEWS для получения полного списка.

Кроме того, был выпущен ряд обновлений для сборки Mac OS X, см. `Mac/BuildScript/README.txt` для получения подробной информации. Для пользователей, использующих 32/64-разрядную сборку, существует известная проблема с Tcl/Tk по умолчанию в Mac OS X 10.6. Соответственно, мы рекомендуем установить обновленную альтернативу, такую как ActiveState Tcl/Tk 8.5.9. См. https://www.python.org/download/mac/tclt / для получения дополнительной информации.

Перенос на Python 3.2

В этом разделе перечислены ранее описанные изменения и другие исправления, которые могут потребовать внесения изменений в ваш код:

- Модуль `configparser` содержит ряд очистительных операций. Основное изменение заключается в замене старого класса `ConfigParser` на давно предпочитаемый альтернативный `SafeConfigParser`. Кроме того, существует ряд более мелких несовместимостей:
  - Синтаксис интерполяции теперь проверяется при выполнении операций `get()` и `set()`. В схеме интерполяции по умолчанию допустимы только два токена со знаками процента: `%(name)s` и `%%`, причем последний является экранированным знаком процента.
  - Методы `set()` и `add_section()` теперь проверяют, что значения являются фактическими строками. Раньше неподдерживаемые типы могли вводиться непреднамеренно.
  - Повторяющиеся разделы или опции из одного источника теперь вызывают либо `DuplicateSectionError`, либо `DuplicateOptionError`. Раньше дубликаты автоматически перезаписывали предыдущую запись.
  - Встроенные комментарии теперь отключены по умолчанию, так что теперь символ `;` можно безопасно использовать в значениях.
  - В комментариях теперь можно делать отступы. Следовательно, для того, чтобы `;` или `#` появлялись в начале строки в многострочных значениях, они должны быть интерполированы. Это предохраняет символы префикса комментария в значениях от того, чтобы их ошибочно принимали за комментарии.
  - `""` теперь является допустимым значением и больше не преобразуется автоматически в пустую строку. Для пустых строк используйте `"option ="` в строке.
- Модуль `nntplib` был сильно переработан, что означает, что его API часто несовместимы с API 3.1.
- Объекты `bytearray` больше нельзя использовать в качестве имен файлов; вместо этого они должны быть преобразованы в байты.
- `array.tostring()` и `array.fromstring()` были переименованы в `array.tobytes()` и `array.frombytes()` для наглядности. Старые названия устарели. (См. bpo-8990.)
- Функции `PyArg_Parse*()`:
  - Формат `"t#"` был удален: вместо него используйте `"s#"` или `"s*"`
  - Форматы `"w"` и `"w#"` были удалены: вместо них используйте `"w*"`
- Тип `PyCObject`, устаревший в версии 3.1, был удален. Чтобы обернуть непрозрачные указатели C в объекты Python, вместо этого следует использовать `PyCapsule API`; новый тип имеет четко определенный интерфейс для передачи информации о безопасности ввода и менее сложную подпись для вызова деструктора.

- Функция `sys.setfilesystemencoding()` была удалена из-за ее некорректного дизайна.
- Функция и метод `random.seed()` теперь засаливают начальные значения строк хэш-функцией `sha512`. Чтобы получить доступ к предыдущей версии `seed` для воспроизведения последовательностей Python 3.1, установите для аргумента `version` значение 1, `random.seed(s, version=1)`.
- Ранее устаревшая функция `string.maketrans()` была удалена в пользу статических методов `bytes.maketrans()` и `bytearray.maketrans()`. Это изменение устраняет путаницу вокруг того, какие типы поддерживались модулем `string`. Теперь `str`, `bytes` и `bytearray` имеют свои собственные методы `maketrans` и `translate` с промежуточными таблицами перевода соответствующего типа.
- Ранее устаревшая функция `contextlib.nested()` была удалена в пользу простого оператора `with`, который может принимать несколько контекстных менеджеров. Последний метод быстрее (потому что он встроенный), и он лучше справляется с завершением нескольких контекстных менеджеров, когда один из них вызывает исключение:

```
with open('mylog.txt') as infile, open('a.out', 'w') as outfile:
     for line in infile:
          if '<critical>' in line:
               outfile.write(line)
```

- `struct.pack()` теперь допускает только байты для кода пакета строк `s`. Раньше он принимал текстовые аргументы и неявно кодировал их в байты, используя UTF-8. Это было проблематично, поскольку делало предположения о правильной кодировке и потому, что кодировка переменной длины может завершиться ошибкой при записи в сегмент структуры фиксированной длины. Код, такой как `struct.pack('<6sHHBBB', 'GIF87a', x, y)`, должен быть переписан с использованием байтов вместо текста, `struct.pack('<6sHHBBB', b'GIF87a', x, y)`.
- Класс `xml.etree.ElementTree` теперь создает `xml.etree.ElementTree.ParseError` при сбое синтаксического анализа. Ранее он вызывал `xml.parsers.expat.ExpatError`.
- Новое, более длинное значение `str()` для чисел с плавающей запятой может привести к нарушению `doctests`, которые полагаются на старый формат вывода.
- В `subprocess.Popen`, значение по умолчанию для `close_fds` теперь равно `True` в Unix; в Windows оно равно `True`, если для трех стандартных потоков установлено значение `None`, в противном случае `False`. Ранее значение `close_fds` по умолчанию всегда было `False`, что приводило к трудноразрешимым ошибкам или условиям гонки, когда дескрипторы открытых файлов просачивались в дочерний процесс.

- Поддержка устаревшего HTTP 0.9 была удалена из `urllib.request` и `http.client`. Такая поддержка все еще присутствует на стороне сервера (в `http.server`).
- Сокеты SSL в режиме тайм-аута теперь вызывают `socket.timeout` при возникновении тайм-аута, а не обычной ошибки SSL.
- Вводящие в заблуждение функции `PyEval_Acquire Lock()` и `PyEval_ReleaseLock()` официально признаны устаревшими. Вместо этого следует использовать API, учитывающие состояние потока (такие как `PyEval_SaveThread()` и `PyEval_RestoreThread()`).
- Из-за угроз безопасности `asyncore.handle_accept()` устарел, и вместо него была добавлена новая функция `asyncore.handle_accepted()`.
- Из-за новой реализации GIL `PyEval_InitThreads()` больше не может быть вызван перед `Py_Initialize()`.