First Edition

INTERMEDIATE PYTHON

Muhammad Yasoob Ullah Khalid

Содержание

Intermediate Python

Python - это невероятный язык с опытным и дружелюбным сообществом программистов. Тем не менее, на данный момент серьезно не хватает информации о хороших источниках, за которые можно было бы взяться после изучения основ языка. Постараюсь решить данную проблему с помощью этой книги. Я предлагаю вам краткое изложение нескольких интересных тем, которые вы сможете позднее изучить самостоятельно в подробностях.

Темы, которые будут затронуты в этой книге, помогут вам открыть для себя множество потайных уголков языка. В конечном итоге я пишу эту книгу как справочник, которое я бы сам хотел иметь, когда начинал свой путь в Python.

Начинающий, продвинутый и даже опытный программист найдет здесь что-нибудь полезное для себя.

Однако, помните что данная книга не представляет из себя руководство и у неё нет цели научить вас Python. Затронутые темы не рассматриваются в подробностях и лишь минимум необходимой информации предлагается вашему вниманию.

Данная книга находится в постоянной доработке. Если вы найдете что-нибудь, что можно было бы улучшить (а я знаю, вам попадется немало таких мест), то отправляйте pull request или открывайте issue.

Скажу даже больше - если вы хотите добавить новые главы, то pull request опять поможет и я буду очень рад расширить книгу.

Уверен, вам уже не терпится также как и мне, так что давайте приступим!

> **Примечание:** Если вы хотите отблагодарить автора за работу, то отличным вариантом будет купить специальную версию книги на Gumroad. Помимо этого, если книга оказалась вам полезной, то поделитесь с автором своим опытом (на английском, пожалуйста). Он будет очень рад прочесть все ваши письма.

Содержание

Скачать / читать книгу

- Онлайн версия на gitbook
- Скачать с gitbook (pdf, mobi, epub)
- Скачать с github (pdf, mobi, epub)

Автор

- Muhammad Yasoob Ullah Khalid

Благодарности

- Philipp Hagemeister:

Написал главу о функции `open`. Спасибо Филипп!

Перевод

Если вы хотите перевести книгу на другой язык - дайте знать автору. Он будет рад расширению аудитории. Список доступных переводов на настоящий момент:

- Английский
- Китайский
- Русский
- Корейский
- Португальский

Отзывы, предложения и помощь

- Как помочь развитию книги
- Оставить свой отзыв

Лицензия

*args и **kwargs

Я много раз замечал, что у новых Python разработчиков вызывают трудности магические переменные *args и **kwargs. Так что же они из себя представляют? Во-первых, позвольте сказать, что вам не обязательно называть их именно *args и **kwargs. Только `*` (звездочка) действительно необходима. К примеру, их можно назвать *var и **vars. Использование *args и **kwargs идет на уровне соглашения между разработчиками. Теперь давайте начнём рассказ с *args.

Использование *args

*args и **kwargs в основном используются в определениях функций. *args и **kwargs позволяют передавать им произвольное число аргументов. Под произвольным числом здесь понимается ситуация, когда вы не знаете заранее сколько аргументов может быть передано функции пользователем, поэтому в данном случае вам необходимо использовать эти ключевые слова. *args используется для передачи произвольного числа **неименованных** аргументов функции. Вот пример, чтобы помочь вам понять идею:

```
def test_var_args(f_arg, *argv):
    print("Первый позиционный аргумент:", f_arg)
    for arg in argv:
        print("Другой аргумент из *argv:", arg)

test_var_args('yasoob', 'python', 'eggs', 'test')
```

Результат будет таким:

```
Первый позиционный аргумент: yasoob
Другой аргумент из *argv: python
Другой аргумент из *argv: eggs
Другой аргумент из *argv: test
```

Надеюсь, пример развеял возможную путаницу. Теперь перейдем к разговору о **kwargs

Использование **kwargs

kwargs позволяет вам передавать произвольное число **именованных аргументов в функцию. Таким образом, вам необходимо использовать **kwargs там, где вы хотите работать с **именованными** аргументами. Очередной пример:

```
def greet_me(**kwargs):
    for key, value in kwargs.items():
        print("{0} = {1}".format(key, value))

>>> greet_me(name="yasoob")
name = yasoob
```

В результате, мы оперируем произвольным числом именованных аргументов в нашей функции. Это были основы использования **kwargs и можете сами отметить насколько это может быть удобно в определенных ситуациях. Теперь давайте рассмотрим использование *args и **kwargs для передачи в функцию списка или словаря аргументов.

Использование *args и **kwargs при вызове функции

Рассмотрим вызов функции, используя *args и **kwargs. Пусть у нас есть вот такая небольшая функция:

```
def test_args_kwargs(arg1, arg2, arg3):
    print("arg1:", arg1)
    print("arg2:", arg2)
    print("arg3:", arg3)
```

Мы можем использовать *args или **kwargs чтобы передать в неё аргументы. Вот как мы это сделаем:

```
# Сначала с *args
>>> args = ("two", 3, 5)
>>> test_args_kwargs(*args)
arg1: two
arg2: 3
arg3: 5

# Теперь с **kwargs:
>>> kwargs = {"arg3": 3, "arg2": "two", "arg1": 5}
>>> test_args_kwargs(**kwargs)
arg1: 5
arg2: two
arg3: 3
```

Порядок использования *args, **kwargs и формальных параметров

Если вы хотите использовать все три метода в функции, то порядок должен быть таким:

```
some_func(fargs, *args, **kwargs)
```

Когда их использовать?

Все зависит от ваших потребностей. Наиболее часто *args и **kwargs используются при написании декораторов (подробнее о декораторах в другой главе). Помимо этого данная методика может использоваться для "monkey patching". Под "monkey patching" понимается модификация кода во время выполнения программы. Предположим, у нас есть класс с методом `get_info`, который обращается к API и возвращает данные ответа. Для тестирования этого метода мы можем заменить вызов API и искусственно возвращать определенные тестовые данные. Например:

```
import someclass

def get_info(self, *args):
    return "Test data"

someclass.get_info = get_info
```

Уверен, вы можете придумать и другие подходящие сценарии использования.

Отладка

Отладка относится к числу навыков, овладев которыми, вы значительно продвинете свои навыки отслеживания багов в коде. Большинство новичков пренебрежительно относятся к важности отладчика Python (`pdb`). В данной главе я расскажу лишь о нескольких важных командах. Узнать больше вы можете из официальной документации.

Запуск из командной строки

Вы можете запустить скрипт из командной строки вместе с отладчиком. Пример:

```
$ python -m pdb my_script.py
```

Отладчик приостановит выполнение программы на первой найденной им инструкции. Это удобно для коротких скриптов. Вы можете проверить значения переменных и продолжить выполнение программы построчно.

Запуск из скрипта

Вы можете задать контрольные точки в коде, что позволит изучить значения переменных и другие параметры в конкретный момент выполнения программы. Это возможно при помощи метода `pdb.set_trace()` . Вот живой пример:

```
import pdb

def make_bread():
    pdb.set_trace()
    return "У меня нет времени"

print(make_bread())
```

Попробуйте запустить этот код. Отладчик откроется сразу после запуска скрипта. Теперь пришло время познакомиться с несколькими командами отладчика.

Команды

- `c` : продолжить выполнения программы
- `w` : отобразить окружение текущей исполняемой инструкции
- `a` : отобразить список аргументов текущей функции
- `s` : исполнить текущую строчку кода и остановиться по возможности
- `n` : продолжить исполнение программы пока не будет достигнута следующая строка текущей функции или пока функция не завершит свою работу

Разница между `n` и `s` в том, что вторая команда приостановит исполнение после вызова функции, а первая только после достижения следующей строки текущей функции.

Это лишь несколько базовых команд. `pdb` также поддерживает проведение анализа после завершения работы программы. Это также очень удобная возможность. Я настоятельно советую вам ознакомиться с официальной документацией и изучить этот инструмент подробнее.

Генераторы

Для начала нам стоит познакомиться с итераторами. Как подсказывает Wiki, итератор — это интерфейс, предоставляющий доступ к элементам коллекции (массива или контейнера). Здесь важно отметить, что итератор только предоставляет доступ, но не выполняет итерацию по ним. Это может звучать довольно запутано, так что остановимся чуть подробнее. Тему итераторов можно разбить на три части:

- Итерируемый объект
- Итератор
- Итерация

Все эти три части связаны друг с другом. Мы обсудим их одну за одной и после перейдем к генераторам.

Итерируемый объект

Итерируемым объектом в Python называется любой объект, имеющий методы `__iter__` или `__getitem__`, которые возвращают **итераторы** или могут принимать индексы. В итоге итерируемый объект это объект, который может предоставить нам **итератор**. Так что же представляет из себя **итератор**?

Итератор

Итератором в Python называется объект, который имеет метод `next` (Python 2) или `__next__`. Вот и все. Это итератор. Теперь об **итерации**.

Итерация

Если коротко - это процесс получения элементов из какого-нибудь источника, например списка. Итерация - это процесс перебора элементов объекта в цикле. Теперь, когда у нас есть общее понимание основных принципов, перейдём к **генераторам**.

Генераторы

Генераторы это итераторы, по которым можно итерировать только один раз. Так происходит поскольку они не хранят все свои значения в памяти, а генерируют элементы "на лету". Генераторы можно использовать с циклом `for` или любой другой функцией или конструкцией, которые позволяют итерировать по объекту. В большинстве случаев **генераторы** создаются как функции. Тем не менее, они не возвращают значение также как функции (т.е. через `return`), в генераторах для этого используется ключевое слово `yield`. Вот простой пример функции-генератора:

```
def generator_function():
    for i in range(10):
        yield i

for item in generator_function():
    print(item)

# Вывод: 0
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

Не самый полезный код, однако достаточно наглядный. Генераторы хорошо подходят для расчета больших наборов результатов (при использовании вложенных циклов), где вам бы не хотелось выделять память для хранения всех результатов одновременно. Многие функции из стандартной библиотеки, возвращающие `списки` в Python 2, были модифицированы, для того, чтобы возвращать `генераторы` в Python 3, поскольку последние требуют меньше ресурсов.

Вот пример `генератора`, который считает числа Фибоначчи:

```
# generator version
def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b
```

А вот так мы можем его использовать:

```
for x in fibon(1000000):
    print(x)
```

С помощью такого метода мы можем не волноваться об использовании большого объема ресурсов. В то же время следующая реализация алгоритма:

```
def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result
```

будет использовать огромный объем наших ресурсов при расчете достаточно больших чисел. Я уже говорил, что мы можем итерировать по **генераторам** только один раз, но давайте проверим это на практике. Перед этим вам надо познакомиться с одной встроенной в язык функцией - `next()` . Она позволяет нам переходить к следующему элементу коллекции. Давайте проверим наше понимание:

```
def generator_function():
    for i in range(3):
        yield i

gen = generator_function()
print(next(gen))
# Вывод: 0
print(next(gen))
# Вывод: 1
print(next(gen))
# Вывод: 2
print(next(gen))
# Вывод: Traceback (most recent call last):
#          File "<stdin>", line 1, in <module>
#        StopIteration
```

Как видно, после прохождения по всем значениям `next()` начала вызывать исключение `StopIteration` . По сути, эта ошибка информирует нас о том, что все значения коллекции уже были пройдены. Может возникнуть вопрос, почему мы не получаем ошибку при использовании цикла `for` . И ответ довольно прост. Цикл `for` автоматически перехватывает данное исключение и перестает вызывать `next` . Знали ли вы, что несколько встроенных типов данных в Python поддерживают итерирование? Давайте посмотрим:

```
my_string = "Yasoob"
next(my_string)
# Вывод: Traceback (most recent call last):
#          File "<stdin>", line 1, in <module>
#        TypeError: str object is not an iterator
```

Ок, это не то что ожидалось. Ошибка говорит, что `str` не итератор. И это действительно так! Строка - итерируемый объект, но не итератор. Т.е. она поддерживает итерирование, но мы не можем делать это напрямую. Так как же нам в конечном итоге итерировать по строке? Пришло время для очередной встроенной функции - `iter` . Она возвращает **итератор** из **итерируемого** объекта. `int` не является итерируемым объектом, однако мы можем использовать `iter` со строками!

```
int_var = 1779
iter(int_var)
# Вывод: Traceback (most recent call last):
#          File "<stdin>", line 1, in <module>
#        TypeError: 'int' object is not iterable
# int не итерируемый объект

my_string = "Yasoob"
my_iter = iter(my_string)
print(next(my_iter))
# Вывод: 'Y'
```

Теперь намного лучше. Я уверен, что вас заинтересовала эта тема. Помните, что полностью изучить генераторы можно только через постоянную практику. Просто используйте **генераторы** везде, где это кажется удачным решением. Вы не разочаруетесь!

Map, Filter и Reduce

В Python есть три функции, которые значительно упрощают функциональный подход к программированию. Мы обсудим их и рассмотрим примеры использования.

Map

`map` применяет функцию ко всем элементам списка. Если коротко:

Сценарий использования

```
map(function_to_apply, list_of_inputs)
```

Нам часто необходимо передать все элементы списка в функцию один за другим и собрать возвращаемые значения в новый список. К примеру:

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
```

`map` позволяет выполнить эту задачу элегантным способом:

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
```

Зачастую с `map` используются анонимные функции, как в примере выше. Вместо списка входных данных можно также использовать список функций!

```
def multiply(x):
    return (x*x)
def add(x):
    return (x+x)

funcs = [multiply, add]
for i in range(5):
    value = list(map(lambda x: x(i), funcs))
    print(value)

# Вывод:
# [0, 0]
# [1, 2]
# [4, 4]
# [9, 6]
# [16, 8]
```

Filter

Как можно догадаться по имени, `filter` возвращает список элементов, для которых заданная функция возвращает `True`. Вот простой и понятный пример:

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)

# Вывод: [-5, -4, -3, -2, -1]
```

`filer` уподобляется циклу, но он является встроенной функцией и работает быстрее.

> **Примечание:** Если `map` и `filter` не кажутся вам достаточно красивым решением, то вы всегда можете использовать абстракции списков/словарей/кортежей. Использование последних считается хорошим тоном, так как практически во всех случаях улучшает читаемость без потери функционала.

Reduce

`Reduce` весьма полезная функция для выполнения вычислений на списке и возвращения единственного результата. Она сворачивает список, применяя полученную в качестве аргумента функцию по очереди к последовательным парам элементов. Например, если мы хотим посчитать произведение всех элементов списка чисел.

Обычным решением этой задачи будет использования цикла `for`:

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num

# product = 24
```

Теперь попробуем с `reduce`:

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])

# Вывод: 24
```

Структура данных `set`

`set` это весьма полезная структура данных. `set` схож по поведению со списком, за тем исключением, что множества неупорядочены и не могут содержать одинаковые значения. Это бывает очень кстати в определённых случаях. К примеру, вам может потребоваться проверить список на наличие дубликатов. Существует два типичных метода. Первый - использовать цикл `for`. Как-то так:

```
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']

duplicates = []
for value in some_list:
    if some_list.count(value) > 1:
        if value not in duplicates:
            duplicates.append(value)

print(duplicates)
# Вывод: ['b', 'n']
```

Но есть и более простое решение при помощи множества. Вы можете сделать что-нибудь такое:

```
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
duplicates = set([x for x in some_list if some_list.count(x) > 1])
print(duplicates)
# Вывод: set(['b', 'n'])
```

У множеств также есть несколько дополнительных методов. Вот некоторые из них:

Пересечение

Вы можете найти пересечение двух множества. Например:

```
valid = set(['yellow', 'red', 'blue', 'green', 'black'])
input_set = set(['red', 'brown'])
print(input_set.intersection(valid))
# Вывод: set(['red'])
```

Разность

Разность двух множеств можно найти следующим методом:

```
valid = set(['yellow', 'red', 'blue', 'green', 'black'])
input_set = set(['red', 'brown'])
print(input_set.difference(valid))
# Вывод: set(['brown'])
```

Множества также можно создавать с помощью новой нотации:

```
a_set = {'red', 'blue', 'green'}
print(type(a_set))
# Вывод: <type 'set'>
```

Существуют и другими методы. Рекомендую ознакомиться с ними в официальной документации.

Тернарные операторы

Тернарные операторы наиболее широко известны в Python как условные выражения. Эти операторы возвращают что-то в зависимости от того, является ли условие истиной или ложью. Они стали частью языка с версии 2.4.

Ниже приведены шаблоны и примеры использования.

Шаблон:

```
condition_is_true if condition else condition_is_false
```

Пример:

```
is_fat = True
state = "fat" if is_fat else "not fat"
```

Такой подход позволяет быстро проверить условие, а не писать несколько строчек оператора `if`. Зачастую это очень удобно, поскольку позволяет писать более компактный код, сохраняя его читабельность.

Другим вариантом (менее очевидным и не настолько широко распространенным) является использование кортежей. Вот пример кода:

Шаблон:

```
(if_test_is_false, if_test_is_true)[test]
```

Пример:

```
fat = True
fitness = ("худой", "толстый")[fat]
print("Али ", fitness)
# Вывод: Али толстый
```

Это работает поскольку в Python `True == 1` и `False == 0`. Помимо кортежей можно использовать списки.

Пример выше редко используется и в основном считается плохой практикой у разработчиков, поскольку не является в должной мере "питонистичным" решением. Вдобавок здесь легко ошибиться в последовательности значений в кортеже.

Другой причиной не пользоваться тернарным оператором на кортежах является обработка всего кортежа при исполнении, когда как для if-else оператора такого не происходит.

Пример:

```
condition = True
print(2 if condition else 1/0)
# Вывод: 2

print((1/0, 2)[condition])
# Было вызвано исключение ZeroDivisionError
```

Во втором примере сначала собирается кортеж, а затем находится элемент под заданным индексом. Тернарный оператор на if-else следует обычной логике условного оператора `if`. Таким образом, если один из случаев может вернуть ошибку или обработка обоих случаев является слишком затратной операцией, то вариант с кортежами точно не стоит использовать.

Декораторы

Декораторы - важная часть Python. Если коротко: они являются функциями, которые изменяют работу других функций. Они помогают делать код короче и более "питонистичным". Большинство новичков не знает где их использовать, так что я расскажу о нескольких случаях, когда декораторы помогут написать лаконичный код.

Для начала рассмотрим как написать свой собственный декоратор.

Это будет самым сложным моментом в теме, поэтому мы будем продвигаться шаг за шагом, так что вы сможете все полностью понять.

Все в Python является объектом

Для начала краткая ретроспектива функций в Python:

```
def hi(name="yasoob"):
    return "Привет " + name

print(hi())
# Вывод: 'Привет yasoob'

# Мы можем присвоить функцию переменной:
greet = hi
# Мы не используем здесь скобки, поскольку наша задача не вызвать функцию,
# а передать её объект переменной. Теперь попробуем запустить

print(greet())
# Вывод: 'Привет yasoob'

# Посмотрим что произойдет, если мы удалим ссылку на оригинальную функцию
del hi
print(hi())
# Вывод: NameError

print(greet())
# Вывод: 'Привет yasoob'
```

Определение функций внутри функций

Итак, это были основы работы с функциями. Теперь продвинемся на шаг дальше. В Python разрешено объявлять функции внутри других функций:

```
def hi(name="yasoob"):
    print("Вы внутри функции hi()")

    def greet():
        return "Вы внутри функции greet()"

    def welcome():
        return "Вы внутри функции welcome()"

    print(greet())
    print(welcome())
    print("Вы внутри функции hi()")

hi()
# Вывод: Вы внутри функции hi()
#        Вы внутри функции greet()
#        Вы внутри функции welcome()
#        Вы внутри функции hi()

# Пример демонстрирует, что при вызове hi() вызываются также функции
# greet() и welcome(). Кроме того, две последние функции недоступны
# извне hi():

greet()
# Вывод: NameError: name 'greet' is not defined
```

Теперь мы знаем, что возможно определять функции внутри других функций. Другими словами: мы можем создавать вложенные функции. Теперь вам нужно познакомиться с еще одной возможностью функций: возвращать другие функции.

Возвращение функции из функции

Нам не обязательно исполнять функцию, определенную внутри другой функции сразу, мы можем вернуть её в качестве возвращаемого значения:

```
def hi(name="yasoob"):
    def greet():
        return "Вы внутри функции greet()"

    def welcome():
        return "Вы внутри функции welcome()"

    if name == "yasoob":
        return greet
    else:
        return welcome

a = hi()
print(a)
# Вывод: <function greet at 0x7f2143c01500>

# Это наглядно демонстрирует, что переменная `a` теперь указывает на
# функцию greet() в функции hi(). Теперь попробуйте вот это

print(a())
# Вывод: Вы внутри функции greet()
```

Давайте еще раз пробежимся по коду. Через условный оператор мы возвращаем из функции объекты `greet` и `welcome`, а не `greet()` и `welcome()`. Почему? Потому что скобки означают вызов функции, без них мы просто передаем сам объект функции. Достаточно ясно? Давайте я чуть подробнее остановлюсь на этом. Когда мы пишем `a = hi()`, функция `hi()` исполняется и (поскольку имя по умолчанию yasoob) возвращается функция `greet`. Если мы изменим код на `a = hi(name="ali")`, то будет возвращена функция `welcome`. Мы также можем набрать `hi()()`, что вернет `Вы внутри функции greet()`.

Передаем функцию в качестве аргумента другой функции

```
def hi():
    return "Привет yasoob!"

def doSomethingBeforeHi(func):
    print("Я делаю что-то скучное перед исполнением hi()")
    print(func())

doSomethingBeforeHi(hi)
# Вывод: Я делаю что-то скучное перед исполнением hi()
#        Привет yasoob!
```

Теперь у нас есть все необходимые знания для изучения работы декораторов. Декораторы позволяют нам исполнять определенный код до и после исполнения конкретной функции.

Пишем наш первый декоратор

В прошлом примере мы по сути уже написали декоратор! Давайте изменим его и сделаем немного более полезным:

```
def a_new_decorator(a_func):

    def wrapTheFunction():
        print("Я делаю что-то скучное перед исполнением a_func()")

        a_func()

        print("Я делаю что-то скучное после исполнения a_func()")

    return wrapTheFunction

def a_function_requiring_decoration():
    print("Я функция, которая требует декорации")

a_function_requiring_decoration()
# Вывод: "Я функция, которая требует декорации"

a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)
# Теперь функция a_function_requiring_decoration обернута в wrapTheFunction()

a_function_requiring_decoration()
# Вывод: Я делаю что-то скучное перед исполнением a_func()
#        Я функция, которая требует декорации
#        Я делаю что-то скучное после исполнения a_func()
```

Все ясно? Мы просто использовали принципы, с которыми познакомились выше. Это то, чем и занимаются декораторы в Python! Они "обертывают" функцию и модифицируют её поведение определенным образом. Сейчас вы можете спросить почему мы не используем в коде символ @. Это просто более короткий способ декорировать функции. Вот как мы можем модифицировать пример выше с использованием @:

```
@a_new_decorator
def a_function_requiring_decoration():
    """Эй ты! Задекорируй меня полностью!"""
    print("Я функция, которая требует декорации")

a_function_requiring_decoration()
# Вывод: Я делаю что-то скучное перед исполнением a_func()
#        Я функция, которая требует декорации
#        Я делаю что-то скучное после исполнения a_func()

# Выражение @a_new_decorator это сокращенная версия следующего кода:
a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)
```

Надеюсь, теперь у вас есть базовое представление о логике работы декораторов в Python. Однако, у нашего кода есть одна проблема. Если мы исполним:

```
print(a_function_requiring_decoration.__name__)
# Вывод: wrapTheFunction
```

Мы этого не ожидали! Имя функции должно быть `a_function_requiring_decoration`. В реальности наша функция была заменена на wrapTheFunction. Она перезаписала имя и строку документации оригинальной функции. К счастью, Python предоставляет нам простой инструмент для обхода этой проблемы - `functools.wraps`. Давайте исправим предыдущий пример, используя `functools.wraps`:

```
from functools import wraps

def a_new_decorator(a_func):
    @wraps(a_func)
    def wrapTheFunction():
        print("Я делаю что-то скучное перед исполнением a_func()")
        a_func()
        print("Я делаю что-то скучное после исполнения a_func()")
    return wrapTheFunction

@a_new_decorator
def a_function_requiring_decoration():
    """Эй ты! Задекорируй меня полностью!"""
    print("Я функция, которая требует декорации")

print(a_function_requiring_decoration.__name__)
# Вывод: a_function_requiring_decoration
```

Так намного лучше. Давайте двигаться дальше и знакомиться с конкретными вариантами использования декораторов.

Макет:

```
from functools import wraps
def decorator_name(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        if not can_run:
            return "Функция не будет исполнена"
        return f(*args, **kwargs)
    return decorated

@decorator_name
def func():
    return("Функция исполняется")

can_run = True
print(func())
# Вывод: Функция исполняется

can_run = False
print(func())
# Вывод: Функция не будет исполнена
```

Примечание: `@wraps` принимает на вход функцию для декорирования и добавляет функциональность копирования имени, строки документации, списка аргументов и т.д. Это открывает доступ к свойствам декорируемой функции из декоратора.

Варианты использования

Теперь давайте рассмотрим области, где декораторы действительно показывают себя и существенно упрощают работу.

Авторизация

Декораторы могут использоваться в веб-приложениях для проверки авторизации пользователя, перед тем как открывать ему доступ к функционалу. Они активно используются в веб-фреймворках Flask и Django. Вот пример проверки авторизации на декораторах:

Пример:

```
from functools import wraps

def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            authenticate()
        return f(*args, **kwargs)
    return decorated
```

Журналирование

Журналирования - другая область, в которой декораторы находя широкое применение. Вот пример:

```
from functools import wraps

def logit(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print(func.__name__ + " была исполнена")
        return func(*args, **kwargs)
    return with_logging

@logit
def addition_func(x):
    """Считаем что-нибудь"""
    return x + x


result = addition_func(4)
# Вывод: addition_func была исполнена
```

Уверен, вы уже думаете о каком-нибудь хитром использовании декораторов.

Декораторы с аргументами

Тогда подумайте вот о чем, является ли `@wraps` также декоратором? Но, `@wraps` же принимает аргумент, как нормальная функция. Тогда почему бы нам не сделать что-то похожее с нашими декораторами?

Когда мы используем синтаксис `@my_decorator` мы применяем декорирующую функцию с декорируемой функцией в качестве параметра. Как вы помните, все в Python является объектом, в том числе и функции! Помня это, мы можем писать

функции, возвращающие декорирующие функции.

Вложенные декораторы внутри функции

Давайте вернемся к нашему примеру с журналированием и напишем декоратор, который позволит нам задавать файл для сохранения логов:

```
from functools import wraps

def logit(logfile='out.log'):
    def logging_decorator(func):
        @wraps(func)
        def wrapped_function(*args, **kwargs):
            log_string = func.__name__ + " была исполнена"
            print(log_string)
            # Открываем логфайл и записваем данные
            with open(logfile, 'a') as opened_file:
                # Мы записываем логи в конкретный файл
                opened_file.write(log_string + '\n')
        return wrapped_function
    return logging_decorator

@logit()
def myfunc1():
    pass

myfunc1()
# Вывод: myfunc1 была исполнена
# Файл out.log создан и содержит строку выше

@logit(logfile='func2.log')
def myfunc2():
    pass

myfunc2()
# Вывод: myfunc2 была исполнена
# Файл func2.log создан и содержит строку выше
```

Декораторы из классов

Теперь наш журналирующий декоратор находится на продакшене, однако, когда отдельные части приложения являются критичными, мы определенно хотим отзываться на возникающие ошибки как можно быстрее. Давайте предположим, что иногда мы просто хотим записывать логи в файл, а иногда мы хотим получать

сообщения об ошибках по email, сохраняя логи в тоже время. Это подходящий случай для использования наследования, однако, до сих пор мы встречали только декораторы-функции.

К счастью, классы также можно использовать для создания декораторов. Давайте опробуем эту методику:

```
class logit(object):
    def __init__(self, logfile='out.log'):
        self.logfile = logfile

    def __call__(self, func):
        log_string = func.__name__ + " была исполнена"
        print(log_string)
        # Открываем логфайл и записваем данные
        with open(self.logfile, 'a') as opened_file:
            # Мы записываем логи в конкретный файл
            opened_file.write(log_string + '\n')
        # Отправляем сообщение
        self.notify()

    def notify(self):
        # Только записываем логи
        pass
```

Такое решение имеет дополнительно преимущество в краткости, в сравнении с вложенными функциями, при этом синтаксис декорирования функции остается прежним:

```
@logit()
def myfunc1():
    pass
```

Теперь давайте возьмем подкласс `logit` и добавим функционал отправки email (эта тема не будет здесь рассмотрена):

```
class email_logit(logit):
    """
    Реализация logit для отправки писем администраторам при вызове
    функции
    """
    def __init__(self, email='admin@myproject.com', *args, **kwargs):
        self.email = email
        super(email_logit, self).__init__(*args, **kwargs)

    def notify(self):
        # Отправляем письмо в self.email
        # Реализация не будет здесь приведена
        pass
```

`@email_logit` будет работать также как и `@logit`, при этом отправляя сообщения на почту администратору помимо журналирования.

Global и Return

Вы, вероятно, встречали функции с ключевым словом `return` в конце. Знаете что оно означает? В целом то же самое что и в других языках. Давайте посмотрим на эту маленькую функцию:

```
def add(value1, value2):
    return value1 + value2

result = add(3, 5)
print(result)
# Вывод: 8
```

Функция выше принимает два значения и возвращает их сумму. Мы также можем переписать её таким образом:

```
def add(value1,value2):
    global result
    result = value1 + value2

add(3, 5)
print(result)
# Вывод: 8
```

Как несложно заметить, мы используем глобальную переменную `result`. Что это означает? К глобальным переменным можно обращаться в том числе и извне области видимости функции. Позвольте продемонстрировать это следующим примером:

```
# Сначала без глобальной переменной
def add(value1, value2):
    result = value1 + value2

add(2, 4)
print(result)

# Вот чёрт, мы наткнулись на исключение. Но почему?
# Интерпретатор Python говорит нам, что не существует
# переменной с именем result. Так произошло, поскольку
# переменная result доступна исключительно из функции,
# где она была определена, если переменная не является
# глобальной.
Traceback (most recent call last):
  File "", line 1, in
    result
NameError: name 'result' is not defined

# Попробуем исправить код, сделав result глобальной
# переменной
def add(value1, value2):
    global result
    result = value1 + value2

add(2, 4)
print(result)
# Вывод: 6
```

Во второй раз ошибок быть не должно. В реальной жизни от глобальных переменных стоит держаться подальше, они только усложняют жизнь, захламляя глобальную область видимости.

Возврат нескольких значений

Что если мы хотим вернуть две переменные из функции вместо одной? Есть несколько способов, которыми пользуются начинающие разработчики. Первый из них - это использование глобальных переменных. Вот подходящий пример:

```
def profile():
    global name
    global age
    name = "Danny"
    age = 30

profile()
print(name)
# Вывод: Danny

print(age)
# Вывод: 30
```

> **Примечание:** как я уже писал, данный метод использовать не рекомендуется. Повторяю, не используйте вышеуказанный метод!

Другим популярным методом является возврат кортежа, списка или словаря с требуемыми значениями.

```
def profile():
    name = "Danny"
    age = 30
    return (name, age)

profile_data = profile()
print(profile_data[0])
# Вывод: Danny

print(profile_data[1])
# Вывод: 30
```

Или общепринятое сокращение:

```
def profile():
    name = "Danny"
    age = 30
    return name, age
```

Это лучший вариант решения проблемы вместе с возвратом списка или словаря. Не используйте глобальные переменные, если точно не уверены в том, что делаете. `global` может быть неплохим вариантом в отдельных редких случаях, но точно не всегда.

Изменяемость

Изменяемые и неизменяемые типы данных в Python традиционно являются причиной головной боли у начинающих разработчиков. Как следует из названия изменяемые объекты *можно модифицировать*, неизменяемые - *постоянны*. Заставим голову кружиться? Смотрим пример:

```
foo = ['hi']
print(foo)
# Вывод: ['hi']

bar = foo
bar += ['bye']
print(foo)
# Вывод: ['hi', 'bye']
```

Что произошло? Мы этого не ожидали! Логично было бы увидеть:

```
foo = ['hi']
print(foo)
# Вывод: ['hi']

bar = foo
bar += ['bye']

print(foo)
# Вывод: ['hi']

print(bar)
# Вывод: ['hi', 'bye']
```

Это не баг, а изменяемые типы данных в действии. Каждый раз, когда вы присваиваете значение переменной изменяемого типа другой переменной, все изменения с этим значением будут отражаться на обоих переменных. Новая переменная становится ссылкой на старую. Так происходит только с изменяемыми типами данных. Вот пример с использованием функций и изменяемых типов данных:

```
def add_to(num, target=[]):
    target.append(num)
    return target

add_to(1)
# Вывод: [1]

add_to(2)
# Вывод: [1, 2]

add_to(3)
# Вывод: [1, 2, 3]
```

Вы могли ожидать другого поведения. Например, что функция `add_to` будет возвращать новый список при каждом вызове:

```
def add_to(num, target=[]):
    target.append(num)
    return target

add_to(1)
# Вывод: [1]

add_to(2)
# Вывод: [2]

add_to(3)
# Вывод: [3]
```

Причиной, опять же, является изменяемость списков, которая и вызывает основную боль. В Python аргументы функции обрабатываются при определении функции, а не при её вызове. По этой причине вы никогда не должны присваивать аргументам по умолчанию значения изменяемых типов, если абсолютно точно не уверены в своих действиях конечно. Правильным подходом будет такой код:

```
def add_to(element, target=None):
    if target is None:
        target = []
    target.append(element)
    return target
```

Каждый раз при вызове функции без аргумента `target` будет создан новый список. К примеру:

```
add_to(42)
# Вывод: [42]

add_to(42)
# Вывод: [42]

add_to(42)
# Вывод: [42]
```

Магия __slots__

В Python каждый класс может иметь атрибуты экземпляров. По умолчанию, Python использует словарь для хранения атрибутов объекта. Это очень удобно, поскольку позволяет добавлять новые атрибуты во время исполнения программы.

Однако, для небольших классов с известными атрибутами такой подход может стать проблемой. Словари занимают большой объём оперативной памяти. Python не может просто выделить заданное количество памяти при создании объекта для хранения его атрибутов. Поэтому большое число объектов будет занимать много оперативной памяти (я говорю о тысячах и миллионах). Тем не менее, существует способ обойти эту проблему. Он включает использование `__slots__`, чтобы Python не создавал словари под хранение атрибутов, а выделял заданный объём памяти для ограниченного числа атрибутов. Вот пример с использованием `__slots__` и без:

Без `__slots__`:

```
class MyClass(object):
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier
        self.set_up()
    # ...
```

С использованием `__slots__`:

```
class MyClass(object):
    __slots__ = ['name', 'identifier']
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier
        self.set_up()
    # ...
```

Второй пример кода уменьшит потребление оперативной памяти. Некоторые люди отмечают 40 и 50% сокращение потребления оперативной памяти при использовании этого решения.

Как вариант, вы может дать шанс PyPy. Он выполняет подобную оптимизацию по умолчанию.

Ниже я привожу пример замера потребления оперативной памяти "с" и без использования `__slots__`, выполненного в IPython, спасибо https://github.com/ianozsvald/ipython_memory_usage

```
Python 3.4.3 (default, Jun  6 2015, 13:32:34)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import ipython_memory_usage.ipython_memory_usage as imu

In [2]: imu.start_watching_memory()
In [2] used 0.0000 MiB RAM in 5.31s, peaked 0.00 MiB above current, total RAM usage 15
.57 MiB

In [3]: %cat slots.py
class MyClass(object):
    __slots__ = ['name', 'identifier']
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier

num = 1024*256
x = [MyClass(1,1) for i in range(num)]
In [3] used 0.2305 MiB RAM in 0.12s, peaked 0.00 MiB above current, total RAM usage 15
.80 MiB

In [4]: from slots import *
In [4] used 9.3008 MiB RAM in 0.72s, peaked 0.00 MiB above current, total RAM usage 25
.10 MiB

In [5]: %cat noslots.py
class MyClass(object):
    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier

num = 1024*256
x = [MyClass(1,1) for i in range(num)]
In [5] used 0.1758 MiB RAM in 0.12s, peaked 0.00 MiB above current, total RAM usage 25
.28 MiB

In [6]: from noslots import *
In [6] used 22.6680 MiB RAM in 0.80s, peaked 0.00 MiB above current, total RAM usage 4
7.95 MiB
```

Виртуальное окружение

Доводилось слышать о `virtualenv`? Если вы начинающий, то, вероятно, нет, для опытного же разработчика это неотъемлемая часть инструментария.

Так что же такое `virtualenv`? `virtualenv` это утилита для создания изолированных окружений в Python. Предположим - у нас есть приложение, которое требует версию 2 определенной библиотеки, однако, другое приложение требует версию 3. Как нам использовать и работать сразу с обоими?

Если вы установите всё необходимое в `/usr/lib/python2.7/site-packages` (путь может отличаться в зависимости от платформы и версии языка), то легко можете случайно обновить уже установленный пакет.

В другом случае, предположим, у нас есть полностью готовое приложение и вы не хотите вносить изменения в его зависимости, в то же время вы работаете над другим приложением, которое требует последних версий библиотек из зависимостей первого приложения.

Что вы будете делать? Использовать `virtualenv`! Оно создаст изолированные окружения для ваших приложений и позволит устанавливать пакеты именно в эти окружения, а не глобально.

Для установки воспользуемся `pip`:

```
$ pip install virtualenv
```

Две основные команды:

- `$ virtualenv myproject`
- `$ source myproject/bin/activate`

Первая создаст новое изолированное окружение в папке `myproject`, а вторая активирует это окружение.

При создании виртуального окружения вам нужно принять решение. Хотите ли вы чтобы виртуальное окружение использовало пакеты, установленные глобально, но по умолчанию у `virtualenv` нет к ним доступа.

Если вы хотите открыть `virtualenv` доступ к глобально установленным пакетам, то используйте флаг `--system-site-packages` во время создания окружения:

```
$ virtualenv --system-site-packages mycoolproject
```

Вы можете деактивировать виртуальное окружение, набрав:

```
$ deactivate
```

Команда `python` вновь будет использовать глобально установленный в системе интерпретатор Python после деактивации виртуального окружения.

Бонус

Вы можете использовать `smartcd` - библиотеку для bash и zsh, с помощью которой автоматически активировать и деактивировать виртуальное окружение при использовании команды `cd`. Больше информации здесь: GitHub

Это было короткое введение в тему виртуальных окружений. Дополнительную информацию можно почерпнуть здесь.

Collections

В стандартную библиотеку Python входит модуль `collections` содержащий дополнительные структуры данных. Мы поговорим о некоторых и обсудим их пользу.

А конкретно:

- `defaultdict`
- `OrderedDict`
- `counter`
- `deque`
- `namedtuple`
- `enum.Enum` (вне модуля; Python 3.4+)

`defaultdict`

Я использую `defaultdict` время от времени. В отличие от `dict` нам не нужно проверять существует ли ключ в словаре или нет. В результате мы можем писать следующий код:

```
from collections import defaultdict

colours = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)

favourite_colours = defaultdict(list)

for name, colour in colours:
    favourite_colours[name].append(colour)

print(favourite_colours)

# Вывод:
# defaultdict(<type 'list'>,
#    {'Arham': ['Green'],
#     'Yasoob': ['Yellow', 'Red'],
#     'Ahmed': ['Silver'],
#     'Ali': ['Blue', 'Black']
# })
```

Другим популярным случаем использования `defaultdict` является добавление элементов в список внутри словаря. Если ключ не существует в словаре, то вы упрётесь в `KeyError`. `defaultdict` позволяет обойти эту проблему аккуратным образом. Для начала, позвольте привести пример использования `dict` с исключением `KeyError`, а затем мы посмотрим на пример с `defaultdict`.

Проблема:

```
some_dict = {}
some_dict['colours']['favourite'] = "yellow"
# Вызывает KeyError: 'colours'
```

Решение:

```
import collections
tree = lambda: collections.defaultdict(tree)
some_dict = tree()
some_dict['colours']['favourite'] = "yellow"
# Работает без ошибок
```

Вы можете вывести в консоль `some_dict` используя `json.dumps`. Вот пример:

```
import json
print(json.dumps(some_dict))
# Вывод: {"colours": {"favourite": "yellow"}}
```

`OrderedDict`

`OrderedDict` сохраняет элементы в порядке добавление в словарь. Изменение значения ключа не изменяет его позиции. При этом удаление и повторное добавление перенесет ключ в конец словаря.

Проблема:

```
colours = {"Red": 198, "Green": 170, "Blue": 160}
for key, value in colours.items():
    print(key, value)
# Вывод:
#   Red 198
#   Blue 160
#   Green 170
#
# Элементы выводятся в произвольном порядке
```

Решение:

```
from collections import OrderedDict

colours = OrderedDict([("Red", 198), ("Green", 170), ("Blue", 160)])
for key, value in colours.items():
    print(key, value)
# Вывод:
#   Red 198
#   Green 170
#   Blue 160
#
# Порядок элементов сохранен
```

`counter`

`Counter` позволяет подсчитывать частоту определенных элементов. К примеру, мы можем использовать его, чтобы посчитать сколько любимых цветов у каждого человека:

```
from collections import Counter

colours = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)

favs = Counter(name for name, colour in colours)
print(favs)
# Вывод: Counter({
#    'Yasoob': 2,
#    'Ali': 2,
#    'Arham': 1,
#    'Ahmed': 1
# })
```

Мы также можем посчитать частоту строк в файле. Пример:

```
with open('filename', 'rb') as f:
    line_count = Counter(f)
print(line_count)
```

deque

`deque` предлагает нам двустороннюю очередь, которая позволяет добавлять и удалять элементы с обеих сторон. Для начала, вам нужно импортировать модуль `deque` из библиотеки `collections`:

```
from collections import deque
```

Теперь мы можем создать экземпляр двусторонней очереди:

```
d = deque()
```

Очередь работает подобно списку в Python и имеет схожие методы. Например, вы можете:

```
d = deque()
d.append('1')
d.append('2')
d.append('3')

print(len(d))
# Вывод: 3

print(d[0])
# Вывод: '1'

print(d[-1])
# Вывод: '3'
```

Мы можем отрезать элементы с обеих сторон очереди:

```
d = deque(range(5))
print(len(d))
# Вывод: 5

d.popleft()
# Вывод: 0

d.pop()
# Вывод: 4

print(d)
# Вывод: deque([1, 2, 3])
```

Мы также можем ограничить число элементов, которые может хранить очередь. Таким образом при достижении максимального числа элементов очередь начнет отрезать элементы с другого конца. Это проще объяснить на примере:

```
d = deque(maxlen=30)
```

Теперь, когда мы попытаемся добавить 31-й элемент - очередь отрежет первый элемент с другого конца. Вы также можете добавлять элементы к очереди с обоих концов:

```
d = deque([1,2,3,4,5])
d.extendleft([0])
d.extend([6,7,8])
print(d)
# Вывод: deque([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

`namedtuple`

Вы уже должны быть знакомы с кортежами. Кортеж в Python это неизменяемый список, который позволяет хранить объекты, разделенные запятой. Они практически идентичны спискам, за исключением нескольких важных особенностей. В отличие от списков, **вы не можете изменить элемент кортежа**. В то же время вы можете обращаться к элементам кортежа по индексам:

```
man = ('Ali', 30)
print(man[0])
# Вывод: Ali
```

Отлично, так что же тогда `namedtuples` ? Этот модуль открывает доступ к удобной структуре данных для простых задач. С помощью именованных кортежей вам не обязательно использовать индексы для обращения к элементам кортежа. Вы можете думать об именованных кортежах как о словарях, но в отличие от словарей они неизменяемы.

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")

print(perry)
# Вывод: Animal(name='perry', age=31, type='cat')

print(perry.name)
# Вывод: 'perry'
```

Теперь вы можете видеть, что мы можем обращаться к элементам именованного кортежа при помощи их имени и `.` (точки). Давайте чуть подробнее на этом остановимся. Именованный кортеж имеет два обязательных аргумента. Это имя самого кортежа и имена полей кортежа. В примере выше имя нашего кортежа `Animal` , имена полей соответственно: `name` , `age` и `type` . Именованный кортеж позволяет создавать **само-документированные** кортежи. Вы сможете легко понять код при первом же взгляде на него. И, поскольку вы не привязаны к индексам, у вас открывается больше возможностей по поддержке своего кода. Помимо этого, **именованные кортежи не создают словари для каждого экземпляра**, они легковесны и не требуют больше памяти чем обычные кортежи. Это делает их быстрее словарей. Тем не менее, помните, что как и в случае с обычными кортежами, **именованный кортеж неизменяем**. Это означает, что такой код работать не будет:

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")
perry.age = 42

# Вывод: Traceback (most recent call last):
#           File "", line 1, in
#        AttributeError: can't set attribute
```

Вы должны использовать именованные кортежи для улучшения читаемости кода. Они **обратносовместимы с обычными кортежами**. Это значит, что вы можете использовать численные индексы с именованными кортежами:

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")
print(perry[0])
# Вывод: perry
```

И последнее, вы можете сконвертировать именованный кортеж в словарь. Вот так:

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="Perry", age=31, type="cat")
print(perry._asdict())
# Вывод: OrderedDict([('name', 'Perry'), ('age', 31), ...
```

`enum.Enum` (Python 3.4+)

Другой полезной структурой данных является `enum`. Он доступен в модуле `enum`, начиная с Python 3.4 (также в PyPI как бекпорт под именем `enum34`). Enums (перечисляемый тип) это простой способ организации разных вещей.

Давайте рассмотрим именованный кортеж `Animal` из прошлого примера. У него есть поле `type`. Проблема в том, что его тип - строка. Это создаёт нам несколько проблем. Что если пользователь ввёл `Cat`, поскольку нажал Shift? Или `CAT`? Или `kitten`?

Перечисление может помочь обойти эту проблему, позволив не использовать строки. Рассмотрим пример:

```
from collections import namedtuple
from enum import Enum

class Species(Enum):
    cat = 1
    dog = 2
    horse = 3
    aardvark = 4
    butterfly = 5
    owl = 6
    platypus = 7
    dragon = 8
    unicorn = 9
    # Список продолжается...

    # Нам безразличен возраст животного, так что мы используем синонимы
    kitten = 1
    puppy = 2

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="Perry", age=31, type=Species.cat)
drogon = Animal(name="Drogon", age=4, type=Species.dragon)
tom = Animal(name="Tom", age=75, type=Species.cat)
charlie = Animal(name="Charlie", age=2, type=Species.kitten)

# А теперь несколько тестов
>>> charlie.type == tom.type
True
>>> charlie.type
<Species.cat: 1>
```

Так у нас куда меньше шансов допустить ошибку. При этом мы должны быть конкретны и использовать только перечисление для определения полей.

Существует три способа получения доступа к перечисляемым элементам. Например, все три метода, представленные ниже, дадут вам значения поля `cat`:

```
Species(1)
Species['cat']
Species.cat
```

Это было короткое погружение в библиотеку `collections`. Обязательно ознакомьтесь с официальной документацией после чтения этой главы.

Enumerate

`enumerate` это встроенная в Python функция. Её пользу не передать одной строкой. В то же время большинство новичков и многие опытные разработчики не знакомы с ней. Она позволяет нам итерировать по объекту с параллельным автоматическим счётчиком. Вот пример:

```
for counter, value in enumerate(some_list):
    print(counter, value)
```

Это не всё! `enumerate` принимает также необязательный аргумент, с помощью которого она становится ещё полезнее.

```
my_list = ['apple', 'banana', 'grapes', 'pear']
for c, value in enumerate(my_list, 1):
    print(c, value)

# Вывод:
# 1 apple
# 2 banana
# 3 grapes
# 4 pear
```

Необязательный аргумент позволяет задавать начальное значение счётчика. Вы также можете создать список кортежей, содержащих индекс и элемент, используя список. Пример:

```
my_list = ['apple', 'banana', 'grapes', 'pear']
counter_list = list(enumerate(my_list, 1))
print(counter_list)
# Вывод: [(1, 'apple'), (2, 'banana'), (3, 'grapes'), (4, 'pear')]
```

Анализ объекта

В программировании, под анализом объекта понимается возможность определения его типа во время исполнения программы. Это одна из сильных сторон Python. Все в Python является объектами и мы можем их исследовать. В языке есть несколько встроенных функций и модулей для этой цели.

`dir`

В этом параграфе мы познакомимся с функцией `dir` и как она помогает нам в анализе объектов.

Это одна из важнейших функций для этой задачи. Она возвращает список атрибутов и методов объекта. Вот пример:

```
my_list = [1, 2, 3]
dir(my_list)
# Вывод: ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
# '__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
# '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
# '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
# '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
# '__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
# '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
# 'remove', 'reverse', 'sort']
```

Мы получили все методы списка. Это может быть полезно, если вы не можете вспомнить имя конкретного метода. Если мы запустим `dir()` без аргументов, то она вернет нам имена всех объектов в текущей области видимости.

`type` и `id`

Функция `type` возвращает тип объекта. Пример:

```
print(type(''))
# Вывод: <type 'str'>

print(type([]))
# Вывод: <type 'list'>

print(type({}))
# Вывод: <type 'dict'>

print(type(dict))
# Вывод: <type 'type'>

print(type(3))
# Вывод: <type 'int'>
```

`id` возвращает уникальный идентификатор объекта. К примеру:

```
name = "Yasoob"
print(id(name))
# Вывод: 139972439030304
```

Модуль `inspect`

Модуль `inspect` также предоставляет несколько полезных функций для получения информации об объектах. Например, вы можете проверить члены объекта, запустив:

```
import inspect
print(inspect.getmembers(str))
# Вывод: [('__add__', <slot wrapper '__add__' of ... ...
```

Существуют и другие методы анализа объектов в Python. Вы можете изучить их отдельно при желании.

Абстракции списков/словарей/множеств

Абстракции - это та особенность Python, которой мне будет очень сильно недоставать, если я сменю язык программирования. Абстракции - это конструкторы, позволяющие создавать последовательности из других последовательностей. В Python (2 и 3) есть три типа подобных абстракций:

- абстракции списков
- абстракции словарей
- абстракции множеств

Мы обсудим все три, хотя освоив абстракции списков вы легко перенесёте знания на остальные типы.

`list` абстракции

Абстракции списков открывают доступ к простому и лаконичному синтаксису генерации списков. Конструкция состоит из квадратных скобок содержащих выражение и оператор `for`, плюс дополнительные `for` или `if` при необходимости. Выражения могут быть любыми, т.е. вам разрешается иметь любые элементы внутри списка. Результатом работы будет новый список после исполнения выражения с оглядкой на `for` и `if`.

Шаблон использования:

```
variable = [out_exp for out_exp in input_list if out_exp == 2]
```

Вот короткий пример:

```
multiples = [i for i in range(30) if i % 3 == 0]
print(multiples)
# Вывод: [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

Такой синтаксис может быть очень удобен для быстрого создания списков. Использование такого подхода вместо функции `filter` позволяет повысить читаемость кода (без потери скорости исполнения). Особенно хорошо абстракции списков заменяют создание нового простого списка при помощи цикла `for` и `append`. Наглядный пример:

```
squared = []
for x in range(10):
    squared.append(x**2)
```

Вы можете сократить решение до одной строки:

```
squared = [x**2 for x in range(10)]
```

`dict` абстракции

Абстракции словарей используются схожим образом. Вот пример, на который я недавно наткнулся:

```
mcase = {'a': 10, 'b': 34, 'A': 7, 'Z': 3}

mcase_frequency = {
    k.lower(): mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0)
    for k in mcase.keys()
}

# mcase_frequency == {'a': 17, 'z': 3, 'b': 34}
```

В примере выше мы суммируем значения ключей, которые отличаются только регистром. Лично я редко пользуюсь этим методом. Вы также можете легко поменять местами ключи и значения в словаре:

```
{v: k for k, v in some_dict.items()}
```

`set` абстракции

Абстракции множеств схожи с абстракциями списков. Единственное различие - используются фигурные скобки `{}`. Вот пример:

```
squared = {x**2 for x in [1, 1, 2]}
print(squared)
# Вывод: {1, 4}
```

Примечание

В русскоязычном сегменте интернета распространение получили другие названия абстракций, в частности используются понятия `генераторы списков/словарей/множеств`. Данные термины, однако, не совсем корректны, так как не являются генераторами (здесь возможна путаница с `genexp`, т.е. `(x**2 for x in [1, 2, 3])` - генераторное выражение, возвращающее объект генератора). В английском используется `list/dict/set comprehension` (`comprehension` - включение). Рекомендуется использовать либо вариант `списковые включения`, либо английское название или его краткую форму `listcomp` / `dictcomp` / `setcomp`.

Исключения

Работа с исключениям это отдельное искусство, освоив которое вы получите инструмент огромного потенциала. Я продемонстрирую несколько методов работы с исключениями в этой главе.

В конечном итоге, нас интересует синтаксис `try/except`. Код, который может вызвать исключение, помещается в `try` блок, обработка исключения - в `except`. Простой пример:

```
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('Было вызвано исключение IOError. {}'.format(e.args[-1]))
```

В примере выше мы перехватываем только исключение `IOError`. Многие новички не знают, что мы можем обрабатывать несколько исключений.

Обработка множества исключений

Мы можем использовать три метода обработки множества исключений. Первый заключается в создании кортежа из всех возможных исключений. Что-то подобное:

```
try:
    file = open('test.txt', 'rb')
except (IOError, EOFError) as e:
    print("Было вызвано исключение. {}".format(e.args[-1]))
```

Другой методы заключается в обработке каждого исключения в отдельном блоке `except`. Мы можем иметь неограниченное их число (но не менее одного). Очередной пример:

```
try:
    file = open('test.txt', 'rb')
except EOFError as e:
    print("Было вызвано исключение EOFError.")
    raise e
except IOError as e:
    print("Было вызвано исключение IOError.")
    raise e
```

Таким образом, если исключение не перехватывается первым блоком `except`, то оно может быть обработано следующим, или не быть обработанным вовсе. Последний метод заключается в перехвате ВСЕХ исключений:

```
try:
    file = open('test.txt', 'rb')
except Exception:
    # Логирование, если оно вам требуется
    raise
```

Это может быть полезно, когда вы не знаете какие исключения могут возникнуть в вашей программе.

`finally`

Основной код помещается в блок `try`. Дальше идут блоки `except`, которые исполняются, если в блоке `try` было вызвано определённое исключение. Третьим типом блоков, следующим за двумя первыми, может быть `finally`. Код в блоке `finally` будет исполнен вне зависимости от того, вызвал ли код в блоке `try` исключение или нет. Это может быть полезно для финальной "чистки" после работы основного скрипта. Вот простой пример:

```
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('Было вызвано исключение IOError. {}'.format(e.args[-1]))
finally:
    print("Я буду напечатан вне зависимости от исключений в блоке try!")

# Вывод: Было вызвано исключение IOError. No such file or directory
#        Я буду напечатан вне зависимости от исключений в блоке try!
```

`try/else`

Иногда мы можем захотеть исполнить определенный код, если исключения **не было**. Это легко сделать с помощью блока `else`. Вы можете спросить: почему нам нужен `else`, если мы можем поместить этот код в блок `try`? Проблема в том, что исключение в этом коде, может быть в свою очередь поймано `try`, а мы можем этого и не хотеть. В целом, `else` нечасто используется, и я, честно говоря, редко к нему прибегаю сам. Пример:

```
try:
    print('Я уверен, исключений не будет!')
except Exception:
    print('Исключение')
else:
    # Любой код, который должен быть исполнен, если исключение в блоке
    # try не было вызвано, но для которого не должна проводиться
    # обработка исключений
    print('Я буду исполнен, если в try не будет исключений.'
          'Мои исключения не будут обрабатываться.')
finally:
    print('Я буду исполнен в любом случае!')

# Вывод: Я уверен, исключений не будет!
#        Я буду исполнен, если в try не будет исключений.
#        Мои исключения не будут обрабатываться.
#        Я буду исполнен в любом случае!
```

Блок `else`, таким образом, исполняется при отсутствии исключений в блоке `try`. `else` исполняется перед `finally`.

Анонимные функции

Анонимные функции это однострочные функции, которые используются в случаях, когда вам не нужно повторно использовать функцию в программе. Они идентичны обыкновенным функциям и повторяют их поведение.

Образец использования

```
lambda argument: manipulate(argument)
```

Пример

```
add = lambda x, y: x + y

print(add(3, 5))
# Вывод: 8
```

Вот несколько случаев, где удобно использовать анонимные функции, и где они часто применяются в реальной жизни:

Сортировка списка

```
a = [(1, 2), (4, 1), (9, 10), (13, -3)]
a.sort(key=lambda x: x[1])

print(a)
# Вывод: [(13, -3), (4, 1), (1, 2), (9, 10)]
```

Параллельная сортировка списков

```
data = zip(list1, list2)
data.sort()
list1, list2 = map(lambda t: list(t), zip(*data))
```

Однострочники

В этой главе я продемонстрирую несколько однострочных команд, которые могут быть очень полезны.

Простой веб-сервер

Когда-нибудь хотели быстро передать файл по сети? Тогда вам повезло. В Python есть такая возможность. Перейдите в директорию, которую хотите расшарить по сети, и наберите следующую команду в терминале:

```
# Python 2
python -m SimpleHTTPServer

# Python 3
python -m http.server
```

Аккуратный вывод в консоль

Вы можете выводить списки и словари аккуратно отформатированными в консоль. Вот нужный код:

```
from pprint import pprint

my_dict = {'name': 'Yasoob', 'age': 'undefined', 'personality': 'awesome'}
pprint(my_dict)
```

Это особенно удобно для словарей. Помимо этого, если вам нужно вывести содержимое JSON файла в терминал в удобочитаемом формате, то:

```
cat file.json | python -m json.tool
```

Профилирование скрипта

Это может быть очень полезно для определения узких мест производительности ваших программ:

```
python -m cProfile my_script.py
```

Примечание: `cProfile` это ускоренная реализация `profile`, написанная на C.

CSV в JSON

Выполните следующую команду:

```
python -c "import csv,json;print json.dumps(list(csv.reader(open('csv_file.csv'))))"
```

Не забудьте заменить `csv_file.csv` на желаемое имя файла.

Сглаживание списка

Вы можете легко и просто сгладить список, содержащий вложенные списки с помощью `itertools.chain.from_iterable` из пакета `itertools`. Вот простой пример:

```
a_list = [[1, 2], [3, 4], [5, 6]]
print(list(itertools.chain.from_iterable(a_list)))
# Вывод: [1, 2, 3, 4, 5, 6]

# или
print(list(itertools.chain(*a_list)))
# Вывод: [1, 2, 3, 4, 5, 6]
```

Однострочные конструкторы

Позволяют избежать больших кусков повторяющегося кода при инициализации класса:

```
class A(object):
    def __init__(self, a, b, c, d, e, f):
        self.__dict__.update({k: v for k, v in locals().items() if k != 'self'})
```

Другие подобные однострочники можно найти на официальном сайте.

For - Else

Циклы - неотъемлемая часть любого языка программирования. В свою очередь цикл `for` также важная часть Python. Однако, существует несколько вещей, связанных с `for`, о которых не знают начинающие разработчики. Мы остановимся в этой главе на паре нюансов.

Начнём с того, что мы уже знаем. Мы можем использовать циклы `for` следующим образом:

```
fruits = ['apple', 'banana', 'mango']
for fruit in fruits:
    print(fruit.capitalize())

# Вывод: Apple
#        Banana
#        Mango
```

Это базовая структура цикла `for`. Теперь перейдем к менее известным особенностям цикла `for` в Python.

`else`

Циклы `for` могут иметь блок `else` и многие не знакомы с этим фактом. Блок `else` выполняется, когда цикл завершается в нормальном режиме. Т.е. не был вызван `break`. Это удобная особенность, которая придется весьма кстати, когда вы поймете где её стоит использовать. Я узнал об этой возможности далеко не сразу.

Типичный пример - поиск элемента в коллекции при помощи цикла `for`. Если элемент найден - мы останавливаем цикл при помощи `break`. Существует два сценария, при которых может завершиться исполнение цикла. Первый - элемент найден и вызван `break`. Второй - элемент так и не был найден и цикл завершился. Мы хотим узнать, какой из этих двух вариантов вызвал остановку цикла. Типичным решением будет создание переменной-флага и её проверка после завершения цикла. Другой способ - использование блока `else`.

Вот решение на `for/else`:

```
for item in container:
    if search_something(item):
        # Нашли!
        process(item)
        break
else:
    # Ничего не найдено...
    not_found_in_container()
```

Сравним с примером из официальной документации:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n/x)
            break
```

Код выше находит целочисленные делители для n. Теперь интересная часть. Мы можем использовать блок `else` чтобы отслеживать простые числа и выводить их на экран:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print( n, 'equals', x, '*', n/x)
            break
    else:
        # Цикл не нашел целочисленного делителя для n
        print(n, 'is a prime number')
```

C-расширения

Интересной возможностью, которую предлагает разработчикам CPython, является простота использования C-кода в Python.

Существует три метода, с помощью которых разработчик может вызвать C функцию из Python кода - `ctypes` , `SWIG` и `Python/C API` . У каждого метода есть свои преимущества и недостатки.

Для начала, зачем нам вообще это может потребоваться?

Несколько популярных причин:

- Вам нужна скорость и вы знаете, что C в 50x раз быстрее Python
- Вам нужна конкретная C-библиотека и вы не хотите писать "велосипед" на Python
- Вам нужен низкоуровневый интерфейс управления ресурсами для работы с памятью и файлами
- Просто потому что Вам так хочется

CTypes

Модуль ctypes один из самых простых способов вызывать C-функции из Python. Он предоставляет C-совместимые типы данных и функции для загрузки DLL, что позволяет обращаться к библиотекам C без их модификации. Отсутствие необходимости изменять C-код объясняет простоту данного метода.

Пример

Простой C-код для суммирования двух чисел, сохраните его как `add.c`

```
// Простой C-файл - суммируем целые и действительные числа

#include <stdio.h>

int add_int(int, int);
float add_float(float, float);

int add_int(int num1, int num2){
    return num1 + num2;
}

float add_float(float num1, float num2){
    return num1 + num2;
}
```

Теперь скомпилируем C-файл в `.so`-файл (DLL под Windows). Так мы получим файл `adder.so`.

```
# Для Linux
$  gcc -shared -Wl,-soname,adder -o adder.so -fPIC add.c

# Для Mac
$ gcc -shared -Wl,-install_name,adder.so -o adder.so -fPIC add.c
```

Теперь Python-код:

```
from ctypes import *

# Загружаем библиотеку
adder = CDLL('./adder.so')

# Находим сумму целых чисел
res_int = adder.add_int(4,5)
print("Сумма 4 и 5 = " + str(res_int))

# Находим сумму действительных чисел
a = c_float(5.5)
b = c_float(4.1)

add_float = adder.add_float
add_float.restype = c_float
print("Сумма 5.5 и 4.1 = " + str(add_float(a, b)))
```

Результат:

```
Сумма 4 и 5 = 9
Сумма 5.5 и 4.1 =  9.60000038147
```

В примере выше, C-файл содержит простой код - две функции: одна для нахождения суммы двух целых чисел, другая - действительных.

В Python-коде мы сначала импортируем модуль `ctypes`. Затем функция `CDLL` из того же модуля используется для загрузки C-библиотеки. Теперь функции из C-кода доступны для нас через переменную `adder`. Когда мы вызываем `adder.add_int()`, то автоматически вызывается C-функция `add_int`. Интерфейс модуля `ctypes` позволяет использовать питоновские целые числа и строки при вызове C-функций.

Для других типов, например логического или действительных чисел, мы должны использовать корректные `ctypes`. Мы делаем это при передаче параметров в `adder.add_float()`. Сначала мы создаём требуемый тип `c_float` из `float`, затем используем в качестве аргумента для C-кода. Этот метод простой и аккуратный, но ограниченный. Мы, к примеру, не можем оперировать объектами на стороне C-кода.

SWIG

Simplified Wrapper and Interface Generator, или SWIG для краткости, это другой способ работы с C-кодом из Python. В этом методе разработчик должен написать отдельный файл, описывающий интерфейс, который будет передаваться в SWIG (утилиту командной строки).

Python-разработчики обычно не используют данный подход, поскольку в большинстве случаев он неоправданно сложен. Тем не менее, это отличный вариант, когда у вас есть C/C++ код, к которому нужно обращаться из множества различных языков.

Пример (с сайта SWIG)

C-код, `example.c` содержит различные функции и переменные:

```
#include <time.h>
double My_variable = 3.0;

int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}

int my_mod(int x, int y) {
    return (x%y);
}

char *get_time()
{
    time_t ltime;
    time(&ltime);
    return ctime(&ltime);
}
```

Файл, описывающий интерфейс. Он не будет изменяться в зависимости от языка, на который вы хотите портировать свой C-код:

```
/* example.i */
%module example
%{
/* Помещаем сюда заголовочные файлы или объявления функций */
extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
%}

extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

Компиляция:

```
unix % swig -python example.i
unix % gcc -c example.c example_wrap.c \
    -I/usr/local/include/python2.1
unix % ld -shared example.o example_wrap.o -o _example.so
```

Python:

```
>>> import example
>>> example.fact(5)
120
>>> example.my_mod(7,3)
1
>>> example.get_time()
'Sun Feb 11 23:01:07 1996'
>>>
```

Как мы можем видеть, SWIG позволяет добиваться нужного нам эффекта, но он требует дополнительных усилий, которые, однако, стоит затратить, если вас интересует возможность запуска C-кода из множества различных языков.

Python/C API

C/Python API это, вероятно, наиболее широко применяемый метод - не благодаря своей простоте, а потому что он позволяет оперировать Python объектами из C кода.

Этот метод подразумевает написание C-кода специально для работы с Python. Все объекты Python представляются как структуры PyObject и заголовочный файл `Python.h` предоставляет различные функции для работы с объектами. Например, если `PyObject` одновременно `PyListType` (список), то мы можем использовать функцию `PyList_Size()`, чтобы получить длину списка. Это эквивалентно коду `len(some_list)` в Python. Большинство основных функций/операторов для стандартных Python объектов доступны в C через `Python.h`.

Пример

Давайте напишем C-библиотеку для суммирования всех элементов списка Python (все элементы являются числами).

Начнем с интерфейса, который мы хотим иметь в итоге. Вот Python-файл, использующий пока отсутствующую C-библиотеку:

```
# Это не простой Python import, addList это C-библиотека
import addList

l = [1,2,3,4,5]
print("Сумма элементов списка - " + str(l) + " = " +  str(addList.add(l)))
```

Смотрится как обыкновенный Python-код, который импортирует и использует Python-модуль `addList`. Единственная разница - модуль `addList` написан на C.

Дальше на повестке у нас C-код, который будет встроен в Python-модуль `addList`, это может смотреться немного странно, однако, разобрав отдельные части, из которых состоит C-файл, вы увидите, что все относительно незатейливо.

adder.c

```
// Python.h содержит все необходимые функции, для работы с объектами Python
#include <Python.h>

// Эту функцию мы вызываем из Python кода
static PyObject* addList_add(PyObject* self, PyObject* args){

  PyObject * listObj;

  // Входящие аргументы находятся в кортеже
  // В нашем случае есть только один аргумент - список, на который мы будем
  // ссылаться как listObj
  if (! PyArg_ParseTuple( args, "O", &listObj))
    return NULL;

  // Длина списка
  long length = PyList_Size(listObj);

  // Проходимся по всем элементам
  int i, sum =0;
  for(i = 0; i < length; i++){
    // Получаем элемент из списка - он также Python-объект
    PyObject* temp = PyList_GetItem(listObj, i);
    // Мы знаем, что элемент это целое число - приводим его к типу C long
    long elem = PyInt_AsLong(temp);
    sum += elem;
  }

  // Возвращаемое в Python-код значение также Python-объект
  // Приводим C long к Python integer
  return Py_BuildValue("i", sum);
}

// Немного документации для `add`
static char addList_docs[] =
    "add( ): add all elements of the list\n";

/*
Эта таблица содержит необходимую информацию о функциях модуля
<имя функции в модуле Python>, <фактическая функция>,
<ожидаемые типы аргументов функции>, <документация функции>
*/
static PyMethodDef addList_funcs[] = {
    {"add", (PyCFunction)addList_add, METH_VARARGS, addList_docs},
    {NULL, NULL, 0, NULL}
};
```

```
/*
addList имя модуля и это блок его инициализации.
<желаемое имя модуля>, <таблица информации>, <документация модуля>
*/
PyMODINIT_FUNC initaddList(void){
    Py_InitModule3("addList", addList_funcs,
                   "Add all ze lists");
}
```

Пошаговое объяснение:

- Заголовочный файл `<Python.h>` содержит все требуемые типы (для представления типов объектов в Python) и определения функций (для работы с Python-объектами).
- Дальше мы пишем функцию, которую собираемся вызывать из Python. По соглашению, имя функции принимается {module-name}_{function-name}, которое в нашем случае - `addList_add`. Подробнее об этой функции будет дальше.
- Затем заполняем таблицу, которая содержит всю необходимую информацию о функциях, которые мы хотим иметь в модуле. Каждая строка относится к функции, последняя - контрольное значение (строка из null элементов).
- Затем идёт блок инициализации модуля - `PyMODINIT_FUNC init{module-name}`.

Функция `addList_add` принимает аргументы типа `PyObject` (args также является кортежем, но поскольку в Python всё является объектами, мы используем унифицированный тип `PyObject`). Мы парсим входные аргументы (фактически, разбиваем кортеж на отдельные элементы) при помощи `PyArg_ParseTuple()`. Первый параметр является аргументом для парсинга. Второй аргумент - строка, регламентирующая процесс парсинга элементов кортежа args. Знак на N-ой позиции строки сообщает нам тип N-ого элемента кортежа args, например - 'i' значит integer, 's' - строка и 'O' - Python-объект. Затем следует несколько аргументов, где мы хотели бы хранить выходные элементы `PyArg_ParseTuple()`. Число этих аргументов равно числу аргументов, которые планируется передавать в функцию модуля и их позиционность должна соблюдаться. Например, если мы ожидаем строку, целое число и список в таком порядке, сигнатура функции будет следующего вида:

```
int n;
char *s;
PyObject* list;
PyArg_ParseTuple(args, "siO", &n, &s, &list);
```

В данном случае, нам нужно извлечь только объект списка и сохранить его в переменной `listObj`. Затем мы используем функцию `PyList_Size()` чтобы получить длину списка. Логика совпадает с `len(some_list)` в Python.

Теперь мы итерируем по списку, получая элементы при помощи функции `PyLint_GetItem(list, index)`. Так мы получаем PyObject*. Однако, поскольку мы знаем, что Python-объекты еще и `PyIntType`, то используем функцию `PyInt_AsLong(PyObj *)` для получения значения. Выполняем процедуру для каждого элемента и получаем сумму.

Сумма преобразуется в Python-объект и возвращается в Python-код при помощи `Py_BuildValue()`. Аргумент "i" означает, что возвращаемое значение имеет тип integer.

В заключение мы собираем C-модуль. Сохраните следующий код как файл `setup.py`:

```
# Собираем модули

from distutils.core import setup, Extension

setup(name='addList', version='1.0',\
      ext_modules=[Extension('addList', ['adder.c'])])
```

и запустите:

```
python setup.py install
```

Это соберёт и установит C-файл в Python-модуль, который нам требуется.

Теперь осталось только протестировать работоспособность:

```
# Модуль, вызывающий C-код
import addList

l = [1,2,3,4,5]
print("Сумма элементов списка - " + str(l) + " = " +  str(addList.add(l)))
```

Результат:

```
Сумма элементов списка - [1, 2, 3, 4, 5] = 15
```

В итоге, как вы можете видеть, мы получили наше первое C-расширение, использующее Python.h API. Этот метод может показаться сложным, однако с практикой вы поймёте его удобство.

Из других методов встраивания C-кода в Python, можно отметить альтернативный и быстрый компилятор Cython. Однако Cython, по сути, отличный от основной ветки Python язык, поэтому я не стал здесь его рассматривать.

Функция `open`

open открывает файл. Логично, правда? Зачастую она используется следующим образом:

```
f = open('photo.jpg', 'r+')
jpgdata = f.read()
f.close()
```

Причина, по которой я пишу эту главу в том, что я очень часто вижу такой код. В нем **три** ошибки. Сможете найти? Если нет - продолжайте читать. В конце главы вы будете знать наверняка, что не так с кодом выше и, что важнее, сможете избегать подобных проблем в своем коде. Давайте начнем с основ.

`open` возвращает дескриптор файла, полученный Python-приложением от вашей операционной системы. Вам требуется вернуть дескриптор назад, после того как работа с файлом завершена, иначе вы можете упереться в ограничение на количество одновременно открытых дескрипторов.

Явно вызывая `close` вы закрываете дескриптор файла, но только при успешном чтении. При вызове Exception после `f = open(...)` `f.close()` не будет выполнен (в зависимости от интерпретатора Python дескриптор может быть возвращён, но это уже другая история). Чтобы быть уверенным в закрытии файла вне зависимости от потенциальных ошибок необходимо использовать выражение `with` :

```
with open('photo.jpg', 'r+') as f:
    jpgdata = f.read()
```

Первый аргумент `open` это файл. Второй - (*метод*) определяет *как* файл будет открыт.

- Если вы хотите прочесть файл - `r`
- Для чтения и записи `r+`
- Для перезаписи содержимого файла `w`
- Добавление информации в файл `a`

Существуют и другие методы, но вы их скорее всего никогда не встретите. Метод важен не только из-за изменения поведения, но и поскольку он может привести к ошибкам доступа. Например, если мы хотим открыть jpg файл в директории,

защищённой от записи, `open(.., 'r+')` вызовет ошибку. Метод также может содержать один дополнительный символ - мы можем открыть файл в бинарном виде (вы получите строку байтов) или в текстовом (строка символов).

В целом, если формат написан людьми, то вам нужен текстовый метод. `jpg` изображения не пишутся строка за строкой людьми (и не могут читаться ими), поэтому их стоит открывать в бинарном виде, добавляя `b` в метод (если вы следуете примеру выше, то корректным будет `rb`). Если вы открываете что-то в текстовом формате (добавьте `t` или ничего к `r/r+/w/a`) вы также должны знать кодировку. Для компьютера все файлы это наборы байтов, не символов.

К сожалению, `open` не позволяет непосредственно выбирать кодировку в Python 2.x. Тем не менее, io.open доступна в обоих ветках Python (в 3.x `open` выступает в качестве алиаса) и делает то, что нам нужно. Вы можете передавать кодировку в аргументе `encoding`. Если вы её не выберете, то система и Python остановятся на кодировке по умолчанию. Вы можете попробовать довериться им, однако стандартный выбор может быть полностью ошибочен или кодировка не сможет отобразить часть символов в файле (такое часто происходит с Python 2.x и Windows). Так что лучше выбирайте её самостоятельно. `utf-8` превосходна и поддерживается большинством браузеров и языков программирования. При записи файла вы можете выбрать любую на свой вкус (или отталкиваясь от предпочтений программы, которая будет этот файл читать).

Как определить кодировку файла, который вы пытаетесь прочесть? К сожалению, нет надежного способа определения правильной кодировки - одни и те же байты могут представлять различные, но разрешенные символы в различных кодировках. По этой причине вам придется опираться на метаданные (например, заголовки HTTP). Все чаще форматы определяют кодировку как UTF-8.

Вооруженные этими знаниями, давайте напишем программу, которая читает файл, определяет является ли он JPG изображением (подсказка: эти файлы начинаются с байтов `FF D8`) и записывает текстовый файл, описывающий входной файл:

```
import io

with open('photo.jpg', 'rb') as inf:
    jpgdata = inf.read()

if jpgdata.startswith(b'\xff\xd8'):
    text = 'Это JPEG файл (%d байт)\n'
else:
    text = 'Это произвольный файл (%d байт)\n'

with io.open('summary.txt', 'w', encoding='utf-8') as outf:
    outf.write(text % len(jpgdata))
```

Теперь, я уверен, вы будете использовать `open` правильно!

Разработка под Python 2+3

Во многих случаях вам может понадобиться писать программы, которые будут корректно работать и на Python 2+ и на 3+.

Представьте, что у вас есть крайне популярный Python-пакет, который используют тысячи людей, но не у всех есть Python 2 или Python 3. В этом случае у вас есть два варианта. Первый - разрабатывать две версии параллельно, одну для второй ветки Python, а другую - для третьей. Другим вариантом будет изменение текущего кода для совместимости как с Python 2, так и с Python 3.

В данном разделе я собираюсь рассказать о нескольких приёмах, которые вы можете использовать, чтобы сделать скрипт совместимым с обоими ветками Python.

Импорты Future

Первым и наиболее важным методом будет использование импорта `__future__`. Это позволяет вам импортировать функционал Python 3 в Python 2. Ниже несколько примеров.

Менеджеры контекста были нововведением в Python 2.6+. Для их использования в Python 2.5 вам необходимо:

```
from __future__ import with_statement
```

`print` стал функцией в Python 3. Для её использования в Python 2 вы можете импортировать функцию из `__future__`:

```
print
# Вывод:

from __future__ import print_function
print(print)
# Вывод: <built-in function print>
```

Переименование модулей

Для начала, скажите мне как вы импортируете модули в ваших скриптах? Большинство делает так:

```
import foo
# или
from foo import bar
```

А знаете ли вы про такой способ:

```
import foo as foo
```

Я знаю, что эффект будет такой же, как и в первом случае, но этот способ критичен для совместимости программ с Python 2 и 3. Попробуем следующий код:

```
try:
    import urllib.request as urllib_request  # для Python 3
except ImportError:
    import urllib2 as urllib_request  # для Python 2
```

Позвольте мне немного пояснить. Мы используем блок `try/except` для импорта. Причина - в Python 2 нет модуля `urllib.request`, поэтому попытка его импорта приведет к `ImportError`. Функциональность `urllib.request` доступна в модуле `urlib2` в Python 2. Таким образом, при использовании Python 2, при импорте `urllib.request` мы упираемся в Exception и импортируем `urllib2` в качестве альтернативы.

Последнее что вам нужно знать - ключевое слово `as`. С его помощью мы импортируем модуль под именем `urllib_request`. Дальше в коде мы просто будем ссылаться на это имя, вне зависимости от того, какую ветку Python мы используем.

Заменяем устаревшие модули Python 2

В Python 2 есть 12 устаревших модулей, которые были удалены в Python 3. Для сохранения совместимости убедитесь, что не используете их в своем коде. Следующим образом можно явно запретить использование удаленного в Python 3 функционала:

```
from future.builtins.disabled import *
```

Теперь, при использовании удаленного в Python 3 модуля, мы будем получать `NameError`:

```
from future.builtins.disabled import *

apply()
# Вывод: NameError: obsolete Python 2 builtin apply is disabled
```

Сторонние бэкпорты

Существует несколько пакетов, которые предоставляют новый функционал Python 3 в Python 2. Например:

- enum `pip install enum34`
- singledispatch `pip install singledispatch`
- pathlib `pip install pathlib`

В качестве дополнительного чтения: официальная документация содержит исчерпывающее руководство, объясняющее шаги, которые вам нужно предпринять для гарантии совместимости кода с обеими ветками Python.

Корутины

Корутины похожи на генераторы за исключением нескольких отличий, основные из которых:

- генераторы возвращают данные
- корутины потребляют данные

Для начала рассмотрим процесс создания генератора. Мы можем создать один таким образом:

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

Такой генератор зачастую используется в цикле `for`:

```
for i in fib():
    print(i)
```

Такой подход отличается скоростью и отсутствием повышенной нагрузки на память, поскольку значения **генерируются** "на лету" и не хранятся в списке. Теперь, если мы используем `yield` в примере выше, то, условно говоря, получим корутину. Корутины потребляют данные, которые им передаются. Вот простой пример реализации `grep` на Python:

```
def grep(pattern):
    print("Searching for", pattern)
    while True:
        line = (yield)
        if pattern in line:
            print(line)
```

Подождите! Что возвращает `yield`? Ну, мы преобразовали её в корутину. Сначала она не содержит значения, вместо этого мы передаём значение из внешнего источника. Мы передаем значения используя метод `.send()`. Вот простой пример:

```
search = grep('coroutine')
next(search)
# Вывод: Searching for coroutine
search.send("I love you")
search.send("Don't you love me?")
search.send("I love coroutines instead!")
# Вывод: I love coroutines instead!
```

Передаваемые значения используются в `yield`. Почему мы вызвали `next()`? Это требуется для запуска корутины. Так же как и в случае с генераторами, корутины не запускают функцию сразу же. Вместо этого они запускают её в ответ на вызов методов `__next__()` и `.send()`. По этой причине вам требуется вызвать `next()`, чтобы исполнение дошло до `yield`.

Мы можем закрыть корутину при помощи метода `.close()`:

```
search = grep('coroutine')
# ...
search.close()
```

Это лишь основы работы с корутинами. Рекомендую дополнительно просмотреть эту шикарную презентацию от David Beazley.

Кэширование функций

Кэширование функций позволяет кэшировать возвращаемые значения функций в зависимости от аргументов. Это может помочь сэкономить время при работе с вводом/выводом на повторяющихся данных. До Python 3.2 мы должны были бы написать собственную реализацию. В Python 3.2+ появился декоратор `lpu_cache`, который позволяет быстро кэшировать возвращаемые функцией значения.

Давайте посмотрим, как мы можем воспользоваться кэшированием в Python 3.2+ и в более старых версиях.

Python 3.2+

Реализуем функцию расчета n-ого числа Фибоначчи с использованием `lru_cache`:

```
from functools import lru_cache

@lru_cache(maxsize=32)
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)

>>> print([fib(n) for n in range(10)])
# Вывод: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Аргумент `maxsize` сообщает `lru_cache` сколько последних значений запоминать.

Мы также можем легко очистить кэш:

```
fib.cache_clear()
```

Python 2+

В старых версиях языка также есть несколько путей достижения схожего эффекта. Вы можете сами реализовать любой тип кэширования. Это зависит только от ваших потребностей. Вот базовое решение:

```
from functools import wraps

def memoize(function):
    memo = {}
    @wraps(function)
    def wrapper(*args):
        if args in memo:
            return memo[args]
        else:
            rv = function(*args)
            memo[args] = rv
            return rv
    return wrapper

@memoize
def fibonacci(n):
    if n < 2: return n
    return fibonacci(n - 1) + fibonacci(n - 2)

fibonacci(25)
```

Отличная статья от Caktus Group, в которой они рассказывают историю бага в Django, который появился из-за `lru_cache`. Рекомендую к ознакомлению.

Менеджеры контекста

Менеджеры контекста позволяют выделять и освобождать ресурсы строго по необходимости. Самый популярный пример использования менеджера контекста - выражение `with`. Предположим, у вас есть две связанные операции, которые вы хотите исполнить в паре, поместив между ними блок кода. Менеджеры контекста позволяют сделать именно это. Например:

```
with open('some_file', 'w') as opened_file:
    opened_file.write('Hola!')
```

Код выше открывает файл, записывает в него данные и закрывает файл после этого. При возникновении ошибки при записи данных в файл менеджер контекста попробует его закрыть. Этот код эквивалентен следующему:

```
file = open('some_file', 'w')
try:
    file.write('Hola!')
finally:
    file.close()
```

Сравнив с первым блоком кода, мы можем заметить замену шаблонного кода на `with`. Основное преимущество использования `with` - это гарантия закрытия файла вне зависимости от того, как будет завершён вложенный код.

Распространенный паттерн использования контекстных менеджеров - блокирование и разблокирование ресурсов, а также закрытие открытых файлов (как я уже показал выше).

Давайте посмотрим, как мы можем написать свой собственный менеджер контекста. Это позволит нам лучше понять логику его работы.

Контекст-менеджер как класс

Необходимый минимум функциональности контекстного менеджера требует методов `__enter__` и `__exit__`. Давайте напишем свой контекстный менеджер для работы с файлами и изучим основы.

```
    class File(object):
        def __init__(self, file_name, method):
            self.file_obj = open(file_name, method)
        def __enter__(self):
            return self.file_obj
        def __exit__(self, type, value, traceback):
            self.file_obj.close()
```

Просто определив методы `__enter__` и `__exit__`, мы можем использовать новый контекстный менеджер с `with`. Давайте попробуем:

```
with File('demo.txt', 'w') as opened_file:
    opened_file.write('Hola!')
```

Метод `__exit__` принимает три аргумента. Они обязательны для любого метода `__exit__` класса контекстного менеджера. Давайте обсудим логику работы:

1. `with` сохраняет метод `__exit__` класса `File`.
2. Следует вызов метода `__enter__` класса `File`.
3. Метод `__enter__` открывает файл и возвращает его.
4. Дескриптор файла передается в `opened_file`.
5. Мы записываем информацию в файл при помощи `.write()`
6. `with` вызывает сохраненный `__exit__` метод.
7. Метод `__exit__` закрывает файл.

Обработка исключений

Мы ещё не успели поговорить об аргументах `type`, `value` и `traceback` метода `__exit__`. Между четвертым и шестым шагом при возникновении исключения, Python передает тип, значение и обратную трассировку исключения методу `__exit__`. Это позволяет методу `__exit__` выбирать способ закрытия файла и выполнять дополнительные действия при необходимости. В нашем случае, мы не уделяем им особого внимания.

Что если объект файла вызвал исключение? Возможно, мы пытаемся вызывать метод на объекте, который его не поддерживает. Например:

```
with File('demo.txt', 'w') as opened_file:
    opened_file.undefined_function('Hola!')
```

Давайте разберём шаги, которые выполняет `with` при возникновении исключения.

1. Тип, значение и обратная трассировка ошибки передается в метод `__exit__` .
2. Обработка исключения передается методу `__exit__`
3. Если `__exit__` возвращает `True` , то исключение было корректно обработано.
4. При возврате любого другого значения `with` вызывает исключение.

В нашем случае метод `__exit__` возвращает `None` (при отсутствии выражения `return` метод в Python возвращает `None`). Таким образом, `with` вызывает исключение.

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AttributeError: 'file' object has no attribute 'undefined_function'
```

Давайте попробуем обработать исключение в методе `__exit__` :

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        print("Исключение было обработано")
        self.file_obj.close()
        return True

with File('demo.txt', 'w') as opened_file:
    opened_file.undefined_function()

# Вывод: Исключение было обработано
```

Наш метод `__exit__` возвращает `True` , таким образом `with` не вызывает исключение.

Это не единственный способ реализации контекстных менеджеров - есть и другой и мы посмотрим на него в следующем параграфе.

Контекст-менеджер из генератора

Мы также можем реализовать менеджер контекста через декораторы и генераторы. В Python присутствует модуль `contextlib` специально для этой цели. Вместо написания класса, мы можем реализовать менеджер контекста из функции-генератора. Посмотрим на простой пример:

```
from contextlib import contextmanager

@contextmanager
def open_file(name):
    f = open(name, 'w')
    yield f
    f.close()
```

Отлично! Реализация менеджера контекста таким способом смотрится более интуитивной и простой. Тем не менее, этот метод требует определённых знаний о генераторах, `yield` и декораторах. В примере выше мы не обрабатываем возможные исключения. В целом, он почти такой же, что и предыдущий.

Давайте чуть подробнее разберем этот подход:

1. Python встречает ключевое слово `yield`. Благодаря этому он создает генератор, а не простую функцию.
2. Благодаря декоратору, `contextmanager` вызывается с функцией `open_file` в качестве аргумента.
3. Функция `contextmanager` возвращает генератор, обёрнутый в объект `GeneratorContextManager`.
4. `GeneratorContextManager` присваивается функции `open_file`. Таким образом, когда мы вызовем функцию `open_file` в следующий раз, то фактически обратимся к объекту `GeneratorContextManager`.

Теперь, когда мы знаем всё это, мы можем использовать созданный менеджер контекста следующим образом:

```
with open_file('some_file') as f:
    f.write('hola!')
```