Несколько полезных паттернов, и что нас ожидает дальше

Паттерн Builder. Создание собственных исключений. Несколько слов о программе Python Advanced.

[Несколько полезных паттернов](#_gphjahwk9c5q)

[Builder](#_ozyx6xbxfsf4)

[Делегирование](#_wk3jpnlu7hda)

[Создание собственных исключений](#_5p58212vnali)

[Несколько слов о следующей части курса](#_8c93t4m30f3w)

[Домашнее задание](#_q81f2g28ili5)

[Дополнительные материалы](#_ms0pceoqro6)

[Используемая литература](#_uvp6qax5r1ok)

# 

# 

# 

# Несколько полезных паттернов

Шаблон проектирования или **паттерн** (англ. design pattern) в разработке программного обеспечения — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

## Builder

**Строитель** (англ. Builder) — порождающий шаблон проектирования предоставляет способ создания составного объекта.

Мы уже работали с JSON-форматом в одном из предыдущих уроков. Предположим, у нас стоит такая задача: загрузить данные о работниках из json-файла и выполнять различные операции с рабочими.

Различные операции с данными было бы удобнее выполнять, работая не словарем, который мы получим загрузив данные из JSON-формата, а с объектом. Напишем простой класс, который на вход будет принимать словарь, а возвращать объект рабочего, автоматически добавляя атрибуты в соответствии с ключами словаря.

|  |
| --- |
| class WorkerBuilder:  def \_\_init\_\_(self, d):  for a, b in d.items():  setattr(self, a, b)  worker = WorkerBuilder({"name": "Петр","surname": "Алексеев","age": 10})  print(worker.name)  print(worker.surname) |

Функция setattr() - добавляет объекту указанный атрибут.

Очень удобно реализовать класс-строитель, принимающий исходные данные любого типа и возвращающий готовый python-объект.

## Делегирование

В ООП также часто используется термин делегирование, под которым обычно подразумевается наличие объекта-контроллера, куда встраиваются другие объекты, получающие запросы на выполнение операций. Контроллеры могут решать административные задачи, такие как слежение за попытками доступа и так далее. В языке Python делегирование часто реализуется с помощью метода \_\_getattr\_\_, потому что он перехватывает попытки доступа к несуществующим атрибутам. Класс-обертка (иногда называется прокси-классом) может использовать метод \_\_getattr\_\_ для перенаправления обращений к обернутому объекту. Класс-обертка повторяет интерфейс обернутого объекта и может добавлять дополнительные операции.

|  |
| --- |
| class Wrapper:  def \_\_init\_\_(self, object):  self.wrapped = object # Сохранить объект  def \_\_getattr\_\_(self, attrname):  print('Trace:', attrname) # Отметить факт извлечения  return getattr(self.wrapped, attrname) # Делегировать извлечение |

Метод \_\_getattr\_\_ получает имя атрибута в виде строки. В этом примере для извлечения из обернутого объекта атрибута, имя которого представлено в виде строки, используется встроенная функция getattr –

Вызов getattr(X, N) аналогичен выражению X.N за исключением того, что N – это выражение, которое во время выполнения представлено строкой, а не именем переменной. Фактически вызов getattr(X, N) по его действию можно сравнить с выражением X.\_\_dict\_\_[N], только в первом случае дополнительно выполняется поиск в дереве наследования, как в выражении X.N, а во втором – нет. Такой приём, реализованный в этом классе-обёртке, можно использовать для управления доступом к любому объекту с атрибутами – спискам, словарям и даже к классам и экземплярам.

|  |
| --- |
| x = Wrapper([1, 2, 3]) # Обернуть список  x.append(4) # Делегировать операцию методу списка  print(x.wrapped) # Вывести обернутый объект  x = Wrapper({'a': 1, 'b': 2}) # Обернуть словарь  print(x.keys()) # Делегировать операцию методу словаря |

Получим вывод:

|  |
| --- |
| Trace: append [1, 2, 3, 4] Trace: keys dict\_keys(['a', 'b']) |

# Создание собственных исключений

Исключение – это своего рода «супер-goto». Обработчик исключений (инструкция try) ставит метку и выполняет некоторый программный код. Если затем где-нибудь в программе возникает исключение, интерпретатор немедленно возвращается к метке, отменяя все активные вызовы функций, которые были произведены после установки метки. Такой подход позволяет соответствующим способом реагировать на необычные события. Кроме того, переход к обработчику исключения выполняется немедленно, поэтому обычно нет никакой необходимости проверять коды возврата каждой вызванной функции, которая могла потерпеть неудачу.

С обработкой исключений мы уже сталкивались, можно, а часто и нужно создавать исключения отличающиеся от стандартных типов. Любое исключение - это экземпляр класса, который должен наследоваться от Exception.

|  |
| --- |
| class ShortInputError(Exception):  """Пользовательский класс исключения."""  def \_\_init\_\_(self, length, atleast):  Exception.\_\_init\_\_(self)  # Добавляем два доп.атрибута, дл вывода более информативного сообщения об ошибке  self.length = length  self.atleast = atleast  # Переопределяем, чтобы вывести сообщение, которое нам нужно  def \_\_str\_\_(self):  return 'ShortInputException: Длина введённой строки -- {0}; ' \  'ожидалось, как минимум, {1}'.format(self.length, self.atleast)  class TestError(ShortInputError):  pass  def max\_char(text):  if len(text) < 2:  # Вызываем исключение с помощью оператора raise  raise ShortInputError(len(text), 2)  # ...  # Перехватываем и обрабатываем свое исключение  try:  max\_char("H")  except ShortInputError as ex:  print(ex)  else:  print('Не было исключений.')  finally:  # Блок finally приведен просто для примера полной структуры перехвата и обработки  pass |

Чтобы явно возбудить исключение, можно использовать инструкцию raise. В общем виде она имеет очень простую форму записи – инструкция raise состоит из слова raise, за которым может следовать имя класса или экземпляр возбуждаемого исключения:

* raise <instance> - возбуждает экземпляр класса-исключения;
* raise <class> - создает и возбуждает экземпляр класса-исключения;
* raise - повторно возбуждает самое последнее исключение.

Как уже упоминалось ранее, исключение в Python – это всегда экземпляр класса. Следовательно, первая форма инструкции raise является наиболее типичной – ей непосредственно передаётся экземпляр класса, который создаётся перед вызовом инструкции raise или внутри неё. Если инструкции передаётся класс, интерпретатор вызовет конструктор класса без аргументов, а полученный экземпляр передаст инструкции raise, если после имени класса добавить круглые скобки, мы получим эквивалентную форму. Третья форма инструкции raise повторно возбуждает текущее исключение – это удобно, когда возникает необходимость передать перехваченное исключение другому обработчику.

Независимо от того, какие исключения будут использованы, они всегда идентифицируются обычными объектами, и только одно исключение может быть активным в каждый конкретный момент времени. Как только исключение перехватывается предложением except, находящемся в любом месте программы, исключение деактивируется (то есть оно не будет передано другой инструкции try), если не будет повторно возбуждено при помощи инструкции raise или в результате ошибки.

# Несколько слов о следующей части курса

На этом мы заканчиваем первую часть курса по python. Мы познакомились с основными инструментами и конструкциями языка, но python содержит ещё много полезных подходов и возможностей.

Во второй части познакомимся с:

* генераторами;
* декораторами;
* множественным наследованием;
* созданием многопоточных приложений;
* с более сложным созданием системных скриптов - незаменимым помощником системных администраторов;
* с работой с базами данных на питоне (на примерах)
* многими другими вещами...

По окончанию двух частей вы сможете с уверенностью считать себя продвинутым python-программистом.

Также, на протяжении последующих вебинаров, будем создавать полноценное приложение, которое можно будет с гордостью прикрепить к своему резюме.

# Домашнее задание

Домашние задания для данного урока вы найдете в папке home\_work в папке с номером текущего урока, скачав проект [здесь](https://github.com/NADiP-Examples/Python_lessons_basic).

Большинство заданий делятся на три категории easy, normal и hard.

* easy - простенькие задачи, на понимание основ;
* normal - если вы делаете эти задачи, то вы хорошо усвоили урок;
* hard - наиболее хитрые задачи, часто с подвохами, для продвинутых слушателей.

Если вы не можете сделать normal задачи - это повод пересмотреть урок, перечитать методичку и обратиться к преподавателю за помощью.

Если не можете сделать hard - не переживайте, со временем научитесь.

Решение большинства задач будем разбирать в начале каждого вебинара.

# Дополнительные материалы

Дополнительные примеры с подробными комментариями находятся [здесь](https://github.com/NADiP-Examples/Python_lessons_basic).

Всё то, о чем сказано здесь, но подробнее:

1. [Пользовательские исключения](http://pythoner.name/documentation/tutorial/exceptions/user-defined)
2. [Паттерны проектирования](http://toly.github.io/blog/2014/03/05/advanced-design-patterns-in-python/)

# 

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Учим python качественно(habr)](https://habrahabr.ru/post/150302/)
2. [Самоучитель по python](http://pythonworld.ru/samouchitel-python)
3. [Книга Лутц М. “Изучаем Python” (4-е издание)](http://www.shashkovs.ru/_prog/Lutc_M._-_Izuchaem_Python_(4-e_izdanie)-_2011.pdf).