10 идиоматических приемов для эффективного программирования на Python

[Jenny V](https://medium.com/@evgenia201002?source=post_page-----28a0b1facc3--------------------------------)

[Jenny V](https://medium.com/@evgenia201002?source=post_page-----28a0b1facc3--------------------------------)

[Dec 27, 2020·12 min read](https://medium.com/nuances-of-programming/10-%D0%B8%D0%B4%D0%B8%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D1%85-%D0%BF%D1%80%D0%B8%D0%B5%D0%BC%D0%BE%D0%B2-%D0%B4%D0%BB%D1%8F-%D1%8D%D1%84%D1%84%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B3%D0%BE-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F-%D0%BD%D0%B0-python-28a0b1facc3?source=post_page-----28a0b1facc3--------------------------------)

https://miro.medium.com/max/60/1*wHRToKuM8RNlKB45XY3smg.png?q=20



Программирование само по себе очень увлекательное занятие, а программирование на Python увлекательнее вдвойне, поскольку в данном языке существует много разных способов реализации одних и тех же функциональностей. Однако в большинстве случаев предпочтительно использовать реализации, присущие исключительно Python, характерной особенностью которых являются чистота и краткость кода.

Программирование на Python или любом другом языке — это не высшая математика, здесь многое зависит от степени сформированности ваших навыков. Если вы целенаправленно начнете писатькод в стиле Python (иначе говоря, питонический), то эти техники вскоре пополнят ваш набор инструментов, и постепенно их использование в проектах станет самым привычным делом. Рассмотрим ряд таких простых приемов, которые, я надеюсь, окажутся вам полезны.

1. Отрицательная индексация

Работать с последовательностями — одно удовольствие, поскольку в таких случаях нам известен порядок элементов, и мы можем с легкостью оперировать ими, опираясь на него. Самыми часто используемыми последовательностями в Python являются строки, кортежи и списки. Доступ к их отдельным элементам происходит с помощью механизма индексации. Подобно другим ведущим языкам программирования Python поддерживает индексацию с отсчётом от 0, при которой мы получаем доступ к первому элементу, используя 0 в квадратных скобках. Помимо этого, можно также задействовать объекты среза для извлечения конкретных элементов последовательности, как показано в нижеприведенных примерах.

>>> # Положительная индексация  
... numbers = [1, 2, 3, 4, 5, 6, 7, 8]  
... print("First Number:", numbers[0])  
... print("First Four Numbers:", numbers[:4])  
... print("Odd Numbers:", numbers[::2])  
...   
First Number: 1  
First Four Numbers: [1, 2, 3, 4]  
Odd Numbers: [1, 3, 5, 7]

Однако Python расширяет свои возможности и поддерживает отрицательную индексацию. Речь идет об использовании -1 для обращения к последнему элементу последовательности и отсчета в обратную сторону. Так предпоследний элемент получает индекс -2 и т. д. Обратите внимание, что отрицательная индексация в объекте среза может также работать с положительным индексом.

>>> # Отрицательная индексация  
... data\_shape = (100, 50, 4)  
... names = ["John", "Aaron", "Mike", "Danny"]  
... hello = "Hello World!"  
...   
... print(data\_shape[-1])  
... print(names[-3:-1])  
... print(hello[1:-1:2])  
...   
4  
['Aaron', 'Mike']  
el ol

2. Проверка пустоты контейнера

Контейнеры относятся к тем типам данных, которые могут хранить в себе другие данные. В число часто используемых встроенных контейнеров входят кортежи, списки, словари и множества. При работе с ними часто приходится проверять, содержат ли они какие-либо элементы, прежде чем приступать к выполнению дополнительных операций. Фактически мы можем проверить длину этих контейнеров, которая соответствует числу сохраненных элементов. Если длина равна 0, то контейнер пуст. Ниже представлен один из простых примеров.

if len(some\_list) > 0:  
 # выполняет определенное действие, если список содержит элементы  
else:  
 # если список пуст - выполняет другое действие

Однако данный способ не совсем питонический. Вместо этого мы можем просто проверить сам контейнер, который в случае наличия в нем элементов будет вычисляться как**True.** Несмотря на то, что следующий пример кода демонстрирует основные контейнеры, такой способ использования применими к строкам (т. е. любые непустые строки — True).

>>> def check\_container\_empty(container):  
... if container:  
... print(f"{container} has elements.")  
... else:  
... print(f"{container} doesn't have elements.")  
...   
... check\_container\_empty([1, 2, 3])  
... check\_container\_empty(set())  
... check\_container\_empty({"zero": 0, "one": 1})  
... check\_container\_empty(tuple())  
...   
[1, 2, 3] has elements.  
set() doesn't have elements.  
{'zero': 0, 'one': 1} has elements.  
() doesn't have elements.

3. Создание списка строк с помощью метода Split()

Строки часто применяются как указатели на определенные объекты. Например, мы можем использовать их для ключей в словаре. В проектах науки о данных строки часто представляют собой имена столбцов. При необходимости выбора нескольких из них мы будем вынуждены создать список строк. Мы действительно можем создавать строки, используя в списке литералы. Однако в этом случае нам придется прописывать пары кавычек, заключающие каждую строку, что для нас, “ленивых” программистов, немного утомительно. В связи с этим я бы предпочел создать список строк, воспользовавшись преимуществом метода split(), как показано в следующем фрагменте кода.

>>> # Список строк   
... # Обычный способ   
... columns = ['name', 'age', 'gender', 'address', 'account\_type']  
... print("\* Literals:", columns)  
...   
... # Вместо этого делаем следующее   
... columns = 'name age gender address account\_type'.split()  
... print("\* Split with spaces:", columns)  
...   
... # Если строки содержат пробелы, то вместо них можно использовать запятые   
... columns = 'name, age, gender, address, account type'.split(', ')  
... print("\* Split with commas:", columns)  
...   
\* Literals: ['name', 'age', 'gender', 'address', 'account\_type']  
\* Split with spaces: ['name', 'age', 'gender', 'address', 'account\_type']  
\* Split with commas: ['name', 'age', 'gender', 'address', 'account type']

Как следует из примера, в качестве разделителей метод split() по умолчанию использует пробелы и создает список строк из строки. Обратите внимание, что, создавая список строк с элементами, содержащими пробелы, вы можете при желании использовать и другой вид разделителя, например запятые.

Такой способ использования возможен благодаря встроенным фукциональностям. К примеру, при создании класса именованного кортежа мы можем сделать следующее: Student = namedtuple(“Student”, [“name”, “gender”, “age”]). Список строк определяет “атрибуты” кортежа. При этом данная операция также по умолчанию осуществима следующим образом: Student = namedtuple(“Student”, “name gender age”). В другом случае создание класса *Enum*поддерживает те же альтернативные решения.

4. Тернарное выражение

Во многих случаях нам нужно определить переменные с конкретными значениями на основе заданных условий, для проверки которых мы можем просто использовать выражение *if…else.*Но для этого требуется несколько строк кода. Если речь идет о присваивании значения одной переменной, то возможен вариант с использованием тернарного выражения, которое проверяет условие и завершает присваивание в одной строке кода. Кроме того, благодаря более краткой форме данного выражения код становится гораздо лаконичнее. Рассмотрим следующий пример:

# Обычный способ   
if score > 90:  
 reward = "1000 dollars"  
else:  
 reward = "500 dollars"# Вместо этого делаем следующее   
reward = "1000 dollars" if score > 90 else "500 dollars"

Иногда данные можно получить из определяемой функции и благодаря этому написать сокращенный вариант операции тернарного выражения, как показано ниже:

# Возможный альтернативный сценарий   
# Вы получили откуда-то премию, но вам не известно является ли она None/0 или нет  
reward = reward\_known or "500 dollars"  
# Вышеуказанная строка кода эквивалентна следующей   
reward = reward\_known if reward\_known else "500 dollars"

5. Инструкция with для файлового объекта

Зачастую нам необходимо как считывать данные из файлов, так и вносить их туда. Самый простой способ — открыть файл, используя встроенную функцию open(), создающую файловый объект, с которым можно работать. Приходилось ли вам ранее решать следующую задачу?

>>> # Создаем текстовый файл с текстом: Hello World!  
...   
... # Открываем файл и добавляем новые данные   
... text\_file0 = open("hello\_world.txt", "a")  
... text\_file0.write("Hello Python!")  
...   
... # Снова открываем файл с какой-либо целью   
... text\_file1 = open("hello\_world.txt")  
... print(text\_file1.read())  
...   
Hello World!

В данном фрагменте кода мы начинаем с текстового файла, содержащего текст “Hello World!”, затем добавляем в него новые данные. Однако через некоторое время, собравшись снова поработать с этим файлом и прочитать из него информацию, мы сталкиваемся с тем, что он по-прежнему содержит старые данные. Иначе говоря, добавленный текст не сохраняется в текстовом файле. Почему же это происходит?

Причина, прежде всего, в том, что мы не закрыли файловый объект, а без этой операции изменения не могут быть сохранены. На самом деле, мы можем напрямую вызвать метод close() для файлового объекта. Однако в нашем распоряжении также есть вариант с использованием инструкции “with”, которая автоматически закроет файловый объект, как показано в примере ниже. Завершив операцию с файлом, убедимся, что он закрыт через обращение к атрибуту файлового объекта closed.

>>> with open("hello\_world.txt", "a") as file:  
... file.write("Hello Python!")  
...   
... with open("hello\_world.txt") as file:  
... print(file.read())  
...   
... print("Is file close?", file.closed)  
...   
Hello World!Hello Python!Hello Python!  
Is file close? True

В более общем смысле инструкция *with —*это синтаксис для использования менеджеров контекста в Python. Предыдущий пример предусматривает операцию с файлом, поскольку файлы являются совместно используемыми ресурсами, и мы несем ответственность за их освобождение. Контекстные менеджеры помогают нам справиться с этой задачей. Как демонстрирует вышеуказанный пример, по завершению операции с файлом он автоматически закрывается через использование инструкции *with.*

6. Вычисление нескольких условий

Довольно часто нам нужно вычислить несколько условий. В связи с этим существует ряд возможных сценариев. Применительно к численным значениям можно использовать несколько сравнений для одной и той же переменной. В этом случае мы можем связать их в цепочку.

# Несколько сравнений   
# Обычный способ   
if a < 4 and a > 1:   
 # выполняем здесь определенное действие # Вместо этого делаем следующее  
if 1 < a < 4:  
 # выполняем здесь определенное действие

В некоторых других сценариях мы сталкиваемся с несколькими сравнениями равенства и используем преимущество следующей техники, проверяя принадлежность с помощью ключевого слова *in*:

# Обычный способ   
if b == "Mon" or b == "Wed" or b == "Fri" or b == "Sun":  
 # выполняем здесь определенное действие # Вместо этого делаем следующее. Также вы можете определить кортеж ("Mon", "Wed", "Fri", "Sun")  
if b in "Mon Wed Fri Sun".split():  
 # выполняем здесь определенное действие

Суть другой техники для вычисления нескольких условий состоит в использовании встроенных функций all() и any(). Говоря точнее, функция all() вычисляется как True, когда все элементы итерируемого объекта — True, в связи с чем она подходит для замены серии логических сравнений AND. С другой стороны, функция any() вычисляется как True, когда любой элемент итерируемого объекта — True, а значит она подходит для замены серии логических операций OR. Обратимся к соответствующим примерам.

# Обычные способы   
if a < 10 and b > 5 and c == 4:  
 # выполняем определенное действиеif a < 10 or b > 5 or c == 4:  
 # выполняем определенное действие# Вместо этого делаем следующее  
if all([a < 10, b > 5, c == 4]):  
 # выполняем определенное действиеif any([a < 10, b > 5, c == 4]):  
 # выполняем определенное действие

7. Использование значений по умолчанию в объявлениях функций

Почти во всех проектах Python большая часть кода предусматривает создание и вызов функций. Иначе говоря, мы постоянно имеем дело с их объявлениями и рефакторингом. Во многих сценариях функцию нужно вызывать несколько раз. В зависимости от различных наборов параметров она будет работать немного по-разному. Однако один набор параметров может использоваться чаще других, и в этом случае при объявлении функций следует рассмотреть установку значений по умолчанию. Обратимся к простому примеру.

# Исходная форма:  
def generate\_plot(data, image\_name):  
 """Эта функция создает диаграмму рассеяния для данных"""  
 # создает диаграмму на основе этихданных   
 ...  
 if image\_name:  
 # сохраняет изображение   
 ...# Во многих случаях нам не нужно сохранять изображение   
generate\_plot(data, None)# Функция со значением по умолчанию   
def generate\_plot(data, image\_name=None):  
 pass# Теперь мы можем опустить второй параметр   
generate\_plot(data)

Примечание. Когда при работе с изменяемыми типами данных (например, списками, множествами) вы устанавливаете значение по умолчанию, убедитесь, что вместо конструктора используете None (например, arg\_name=[]). Так как Python создает объект функции там, где он определен, предоставление пустого списка будет за ним “закреплено”. Иначе говоря, при вызове объекта функции, этот объект не будет создаваться попутно. Вместо этого вы будете иметь дело с тем же объектом функции, включая его предустановленный изменяемый объект, изначально созданный в памяти, что может привести к неожиданному поведению.

8. Использование счетчика для подсчета элементов

Работая со списками, кортежами или строками, содержащими несколько элементов (например, несколько символов), у нас часто возникает потребность подсчитать количество каждого из них. Для реализации данной функциональности можно написать вот такой громоздкий код:

>>> words = ['an', 'boy', 'girl', 'an', 'boy', 'dog', 'cat', 'Dog', 'CAT', 'an','GIRL', 'AN', 'dog', 'cat', 'cat', 'bag', 'BAG', 'BOY', 'boy', 'an']  
... unique\_words = {x.lower() for x in set(words)}  
... for word in unique\_words:  
... print(f"\* Count of {word}: {words.count(word)}")  
...   
\* Count of cat: 3  
\* Count of bag: 1  
\* Count of boy: 3  
\* Count of dog: 2  
\* Count of an: 5  
\* Count of girl: 1

Как показано выше, сначала мы должны создать множество, включающее только уникальные слова. После этого мы перебираем это множество слов и используем метод count(), чтобы установить количество вхождений каждого из них. Однако есть более эффективный способ для решения этой задачи — применение класса Counter, предназначенного для подсчета элементов.

>>> from collections import Counter  
...   
... word\_counter = Counter(x.lower() for x in words)  
... print("Word Counts:", word\_counter)  
...   
Word Counts: Counter({'an': 5, 'boy': 4, 'cat': 4, 'dog': 3, 'girl': 2, 'bag': 2})

Класс *Counter*доступен в модуле collections. Для него мы просто создали генератор x.lower() for x in words и произвели подсчет каждого из элементов. Как следует из примера, *Counter*является подобным словарю объектом отображения, где каждый ключ соответствует уникальному элементу списка слов, а значения представляют собой подсчитанное количество этих элементов. Довольно лаконично, не так ли?

Более того, если вам понадобится выявить наиболее часто встречающиеся в списке слов элементы, можете воспользоваться методом most\_common() объекта *Counter.*В следующем примере кода демонстрируются особенности его применения. Вам лишь необходимо указать целое число (N), которое позволит определить N наиболее часто встречающихся элементов в списке. Следует также отметить, что объект Counter будет работать и с другими последовательностями, такими как строки и кортежи.

>>> # Найти наиболее часто встречающийся элемент   
... print("Most Frequent:", word\_counter.most\_common(1))  
Most Frequent: [('an', 5)]  
>>> # Найти 2 наиболее встречающихся элемента   
... print("Most Frequent:", word\_counter.most\_common(2))  
Most Frequent: [('an', 5), ('boy', 4)]

9. Сортировка согласно различным условиям последовательности

Во многих проектах перед нами стоит задача сортировки элементов списка. В основном ее проводят по числовому или алфавитному порядку, используя функцию sorted(). По умолчанию данная функция сортирует список (на самом деле, это может быть любой итерируемый объект) по возрастанию. Если мы зададим условие, при котором аргумент reverse является True, то сможем получить последовательность элементов в порядке убывания. Обратимся к простым примерам.

>>> # Список чисел и строк   
... numbers = [1, 3, 7, 2, 5, 4]  
... words = ['yay', 'bill', 'zen', 'del']  
... # Сортируем их   
... print(sorted(numbers))  
... print(sorted(words))  
...   
[1, 2, 3, 4, 5, 7]  
['bill', 'del', 'yay', 'zen']  
>>> # Сортируем их в порядке убывания   
... print(sorted(numbers, reverse=True))  
... print(sorted(words, reverse=True))  
...   
[7, 5, 4, 3, 2, 1]  
['zen', 'yay', 'del', 'bill']

Помимо основных случаев использования, можно указать аргумент key для сортировки сложных элементов, таких как список кортежей. Рассмотрим соответствующий пример.

>>> # Создаем список кортежей   
... grades = [('John', 95), ('Aaron', 99), ('Zack', 97), ('Don', 92), ('Jennifer', 100), ('Abby', 94), ('Zoe', 99), ('Dee', 93)]  
>>> # Сортируем по оценкам в порядке убывания   
... sorted(grades, key=lambda x: x[1], reverse=True)  
[('Jennifer', 100), ('Aaron', 99), ('Zoe', 99), ('Zack', 97), ('John', 95), ('Abby', 94), ('Dee', 93), ('Don', 92)]  
>>> # Сортируем по начальной букве имени в порядке возрастания   
... sorted(grades, key=lambda x: x[0][0])  
[('Aaron', 99), ('Abby', 94), ('Don', 92), ('Dee', 93), ('John', 95), ('Jennifer', 100), ('Zack', 97), ('Zoe', 99)]

В данном коде показаны 2 продвинутых примера сортировки с использованием лямбда-функции, которая передается в аргумент key. В первом случае элементы сортируются в порядке убывания, а во втором — в установленном по умолчанию порядке возрастания. А что если объединить два этих условия? Если вы рассматриваете возможность применения аргумента reverse, то, скорее всего, вас ждет разочарование, поскольку при попытке провести сортировку по нескольким критериям аргумент reverse будет применен ко всем. Что же предпринять? Обратимся к следующему фрагменту кода.

>>> # Условие: сортировка по начальной букве имени в порядке возрастания и по оценкам в порядке убывания   
... # Оба способа не сработают   
... sorted(grades, key=lambda x: (x[0][0], x[1]), reverse=True)  
[('Zoe', 99), ('Zack', 97), ('Jennifer', 100), ('John', 95), ('Dee', 93), ('Don', 92), ('Aaron', 99), ('Abby', 94)]  
>>> sorted(grades, key=lambda x: (x[0][0], x[1]), reverse=False)  
[('Abby', 94), ('Aaron', 99), ('Don', 92), ('Dee', 93), ('John', 95), ('Jennifer', 100), ('Zack', 97), ('Zoe', 99)]  
>>> # Этот прием решит задачу   
... sorted(grades, key=lambda x: (x[0][0], -x[1]))  
[('Aaron', 99), ('Abby', 94), ('Dee', 93), ('Don', 92), ('Jennifer', 100), ('John', 95), ('Zoe', 99), ('Zack', 97)]

Как вы видите, ни один из вариантов установки аргумента reverse, будь то True или False, не сработал. А вот прием с отрицанием оценок оказался успешным, так как при предустановленной сортировке по возрастанию баллы будут отсортированы в обратном порядке вследствие отрицания этих значений. Однако, прибегая к этому методу, следует помнить, что отрицания применимы только к числовым значениям, но не к строкам.

10. Прием с defaultdict

Словари являются эффективным типом данных, позволяющим хранить их в виде пар ключ-значение. Все ключи изначально должны быть хэшируемыми, поскольку хранение этих данных может подразумевать использование хэш-таблицы. Такая реализация позволяет добиться эффективности O(1) при извлечении и вводе данных. Однако следует отметить, что кроме встроенного типа *dict*мы можем использовать альтернативные словари, на один из которых, а именно тип *defaultdict,*мне бы хотелось обратить ваше внимание. *Defaultdict,*в отличие от встроенного типа *dict,*позволяет нам по умолчанию установить фабричную функцию, которая в случае отсутствия ключа создает элемент. Вам, вероятно, приходилось встречать ошибку такого рода:

>>> student = {'name': "John", 'age': 18}  
... student['gender']  
...   
Traceback (most recent call last):  
 File "<input>", line 2, in <module>  
KeyError: 'gender'

Предположим, что мы имеем дело со словами, и нам нужно сгруппировать одинаковые символы в список. Эти списки соотносятся с символами, являющимися ключами. Приведем пример простой реализации с использованием встроенного типа *dict.*При этом необходимо проверить, имеет ли объект dict ключ letter, поскольку, если его не существует, то вызов метода append() может привести к исключению KeyError.

>>> letters = ["a", "a", "c", "d", "d", "c", "a", "b"]  
... final\_dict = {}  
... for letter in letters:  
... if letter not in final\_dict:  
... final\_dict[letter] = []  
... final\_dict[letter].append(letter)  
...   
... print("Final Dict:", final\_dict)  
...   
Final Dict: {'a': ['a', 'a', 'a'], 'c': ['c', 'c'], 'd': ['d', 'd'], 'b': ['b']}

Теперь рассмотрим вариант с *defaultdict*для написания более краткого кода. Несмотря на свою простоту, предложенный пример позволит вам составить представление о классе *defaultdict,*который избавляет нас от необходимости работать с несуществующими среди объектов словаряключами .

>>> from collections import defaultdict  
...   
... final\_defaultdict = defaultdict(list)  
... for letter in letters:  
... final\_defaultdict[letter].append(letter)  
...   
... print("Final Default Dict:", final\_defaultdict)  
...   
Final Default Dict: defaultdict(<class 'list'>, {'a': ['a', 'a', 'a'], 'c': ['c', 'c'], 'd': ['d', 'd'], 'b': ['b']})

Заключение

Возможно, какие-то приемы были вам известны и до знакомства с материалом данной статьи, но надеюсь, что вы провели время с пользой и освежили свои знания. Использование питонических техник в проектах поможет вам писать более читаемый и производительный код.