Встроенные функции Python: какие нужно просто знать

В Python существуют десятки встроенных функций и классов, сотни инструментов, входящих в стандартную библиотеку Python, и тысячи сторонних библиотек на PyPI. Держать всё в голове начинающему программисту нереально. В статье расскажем про стандартные встроенные функции Python: какие используются часто, а какие вам, вероятно, не пригодятся никогда.

Чтобы разобраться, на какие функции стоит обратить внимание, их следует разделить на группы:

- ***общеизвестные***: почти все новички используют эти функции и довольно часто (print, len, str, int, float, list, tuple, dict, set, range);
- ***неочевидные для новичков***: об этих функциях полезно знать, но их легко пропустить, когда вы новичок в Python (bool, enumerate, zip, reversed, sum, min и max, sorted, any и all);
- ***понадобятся позже***: об этих встроенных функциях полезно помнить, чтобы найти их потом, когда/если они понадобятся (open, input, repr, super, property, issubclass и isinstance, hasattr, getattr, setattr и delattr, classmethod и staticmethod, next);
- ***можно изучить когда-нибудь***: это может пригодиться, но только при определённых обстоятельствах;
- ***скорее всего, они вам не нужны***: они вряд ли понадобятся, если вы не занимаетесь чем-то достаточно специализированным.

Встроенные функции в первых двух категориях являются основными. Они в конечном итоге будут нужны почти всем начинающим программистам на Python. Встроенные модули в следующих двух категориях являются специализированными, но потребности в них будут варьироваться в зависимости от вашей

специализации. Категория 5 — это скрытые встроенные функции. Они очень полезны, когда в них есть необходимость, но многим программистам Python они, вероятно, никогда не понадобятся.

Общеизвестные функции

Если вы уже писали код на Python, эти модули должны быть вам знакомы.

print

Вряд ли найдётся разработчик, даже начинающий, который не знает эту функцию вывода. Как минимум реализация «Hello, World!» требует использования данного метода. Но вы можете не знать о различных аргументах, которые можно передавать в print.

```
>>> words = ["Welcome", "to", "Python"]
>>> print(words)
['Welcome', 'to', 'Python']
>>> print(*words, end="!\n")
Welcome to Python!
>>> print(*words, sep="\n")
Welcome
to
Python
```

len

В Python нет синтаксиса вроде my_list.length() или my_string.length, вместо этого используются поначалу непривычные конструкции len(my_list) и len(my_string).

```
>>> words = ["Welcome", "to", "Python"]
>>> len(words)
```

Нравится вам такая реализация или нет, другой альтернативы не предусмотрено, поэтому к ней нужно привыкнуть.

str

К сожалению, в отличие от многих других языков программирования, в Python нельзя объединять строки и числа.

```
>>> version = 3
>>> "Python " + version
Traceback (most recent call last):
  File "", line 1, in
TypeError: can only concatenate str (not "int") to str
```

Python отказывается приводить целое число 3 к типу строка, поэтому нужно сделать это самостоятельно, используя встроенную функцию str (технически это класс, но с целью уменьшить количество ненужной информации будем принимать все методы за функции).

```
>>> version = 3
>>> "Python " + str(version)
'Python 3'
```

int

Если нужно пользовательский ввод преобразовать в integer, эта функция незаменима. Она может преобразовывать строки в целые числа.

```
>>> program_name = "Python 3"
>>> version_number = program_name.split()[-1]
>>> int(version_number)
3
```

Эту функцию также можно использовать для отсечения дробной части у числа с плавающей точкой.

```
>>> from math import sqrt
>>> sqrt(28)
5.291502622129181
```

```
>>> int(sqrt(28))
5
```

Обратите внимание, если нужно обрезать дробную часть при делении, оператор «//» более уместен (с отрицательными числами это работает иначе).

```
int (3/2) == 3 // 2
```

float

Если строка, которую надо конвертировать в число, не является целым числом, здесь поможет метод float.

```
>>> program_name = "Python 3"
>>> version_number = program_name.split()[-1]
>>> float(version_number)
3.0
>>>                          pi_digits                          =
'3.14159265358979323846264338327950288419716939937 5'
>>> len(pi_digits)
50
>>> float(pi_digits)
3.141592653589793
```

Float также можно использовать для преобразования целых чисел в числа с плавающей запятой.

В Python 2 такое преобразование необходимо, но в Python 3 целочисленное деление больше не является чем-то особенным (если вы специально не используете оператор «//»). Поэтому больше не нужно использовать float для этой цели, теперь float(x)/y можно легко заменить на x/y.

list

Эта функция может очень облегчить задачу, если вы хотите составить список из итераций цикла.

```
>>> numbers = [2, 1, 3, 5, 8]
>>> squares = (n**2 for n in numbers)
>>> squares
 at 0x7fd52dbd5930>
>>> list_of_squares = list(squares)
>>> list_of_squares
[4, 1, 9, 25, 64]
```

При работе со списком метод copy позволяет создать его копию.

```
>>> copy_of_squares = list_of_squares.copy()
```

Если вы не знаете, с какими элементами работаете, функция list является более общим способом перебора элементов и их копирования.

```
>>> copy_of_squares = list(list_of_squares)
```

Также можно использовать списковое включение, но делать это не рекомендуется.

Обратите внимание, когда вы хотите создать пустой список, следует использовать буквальный синтаксис списка («[ ]»).

```
>>> my_list = list()  # Так делать нельзя
>>> my_list = []      # Так можно
```

Использование «[ ]» считается более идиоматическим, так как эти скобки на самом деле выглядят как список Python.

tuple

Эта функция во многом похожа на функцию list, за исключением того, что вместо списков она создает кортежи.

```
>>> numbers = [2, 1, 3, 4, 7]
>>> tuple(numbers)
(2, 1, 3, 4, 7)
```

Если вы пытаетесь создать хешируемую коллекцию (например, ключ словаря), стоит отдать предпочтению кортежу вместо списка.

dict

Эта функция создаёт новый словарь. Подобно спискам и кортежам, dict эквивалентна проходу по массиву пар «ключ-значение» и созданию из них словаря. Дан список кортежей, по два элемента в каждом.

```
>>> color_counts = [('red', 2), ('green', 1), ('blue', 3), ('purple', 5)]
```

Выведем его на экран с помощью цикла.

```
>>> colors = {}
>>> for color, n in color_counts:
...     colors[color] = n
...
>>> colors
{'red': 2, 'green': 1, 'blue' 3, 'purple': 5}>
```

То же самое, но с использованием dist.

```
>>> colors = dict(color_counts)
>>> colors
{'red': 2, 'green': 1, 'blue' 3, 'purple': 5}>
```

Функция dist может принимать 2 типа аргументов:

- другой словарь: в этом случае этот словарь будет скопирован;
- список кортежей «ключ-значение»: в этом случае из этих слов будет создан новый словарь.

Поэтому следующий код также будет работать.

```
>>> colors
```

```
{'red': 2, 'green': 1, 'blue' 3, 'purple': 5}
>>> new_dictionary = dict(colors)
>>> new_dictionary
{'red': 2, 'green': 1, 'blue' 3, 'purple': 5}
```

Функция dict также может принимать ключевые слова в качестве аргументов для создания словаря со строковыми ключами.

```
>>> person = dict(name='Trey Hunner', profession='Python Trainer')
>>> person
{'name': 'Trey Hunner', 'profession': 'Python Trainer'}
```

Но рекомендуется всё же использовать литералы вместо ключевых слов.

```
>>> person = {'name': 'Trey Hunner', 'profession': 'Python Trainer'}
>>> person
{'name': 'Trey Hunner', 'profession': 'Python Trainer'}
```

Такой синтаксис более гибок и немного быстрее. Но самое главное он более чётко передаёт факт того, что вы создаёте именно словарь.

Как в случае со списком и кортежем, пустой словарь следует создавать с использованием буквального синтаксиса («{}»).

```
>>> my_list = dict()  # Так делать нельзя
>>> my_list = {}  # Так можно
```

Использование «{}» более идиоматично и эффективно с точки зрения использования процессора. Обычно для создания словарей используются фигурные скобки, dict встречается гораздо реже.

set

Функция set создаёт новый набор. Она принимает итерации из хешируемых значений (строк, чисел или других неизменяемых

типов) и возвращает set.

```
>>> numbers = [1, 1, 2, 3, 5, 8]
>>> set(numbers)
{1, 2, 3, 5, 8}
```

Создать пустой набор с «{}» нельзя (фигурные скобки создают пустой словарь). Поэтому функция set — лучший способ создать пустой набор.

```
>>> numbers = set()
>>> numbers
set()
```

Можно использовать и другой синтаксис.

```
>>> {*()}  # Так можно создать пустой набор
set()
```

Такой способ имеет недостаток — он сбивает с толку (он основан на редко используемой функции оператора *), поэтому он не рекомендуется.

range

Эта функция создаёт объект range, который представляет собой диапазон чисел.

```
>>> range(10_000)
range(0, 10000)
>>> range(-1_000_000_000, 1_000_000_000)
range(-1000000000, 1000000000)
```

Результирующий диапазон чисел включает начальный номер, но исключает конечный (range(0, 10) не включает 10). Данная функция полезна при переборе чисел.

```
>>> for n in range(0, 50, 10):
...     print(n)
...
0
10
20
30
40
```

Обычный вариант использования – выполнить операцию n раз.

```
first_five = [get_things() for _ in range(5)]
```

Функция range в Python 2 возвращает список. Это означает, что примеры кода выше будут создавать очень большие списки. Range в Python 3 работает как xrange в Python 2. Числа вычисляются «более лениво» при проходе по диапазону.

Функции, неочевидные для новичков

bool

Эта функция проверяет достоверность (истинность) объектов Python. Относительно чисел будет выполняться проверка на неравенство нулю.

```
>>> bool(5)
True
>>> bool(-1)
True
>>> bool(0)
False
```

Применяя bool к коллекциям, будет проверяться их длина (больше 0 или нет).

```
>>> bool('hello')
True
```

```
>>> bool('')
False
>>> bool(['a'])
True
>>> bool([])
False
>>> bool({})
False
>>> bool({1: 1, 2: 4, 3: 9})
True
>>> bool(range(5))
True
>>> bool(range(0))
False
>>> bool(None)
False
```

Проверка истинности очень важна в Python. Вместо того, чтобы задавать вопросы о длине контейнера, многие новички задают проверку истинности.

```
# Вместо этого
if len(numbers) == 0:
    print("The numbers list is empty")

# многие делают так
if not numbers:
    print("The numbers list is empty")
```

Данная функция используется редко. Но, если нужно привести значение к логическому типу для проверки его истинности, bool вам необходима.

enumerate

Если нужно в цикле посчитать количество элементов (по одному элементу за раз), эта функция может быть очень полезной. Такая задача может показаться специфической, но она бывает нужна довольно часто. Например, если нужно отслеживать номер строки в файле.

```
>>> with open('hello.txt', mode='rt') as my_file:
...     for n, line in enumerate(my_file, start=1):
...         print(f"{n:03}", line)
...
001 This is the first line of the file
002 This is the second line
003 This is the last line of the file
```

Enumerate также часто используется для отслеживания индекса элементов в последовательности.

```
def palindromic(sequence):
     """Возвращает True, если последовательность является палиндромом."""
    for i, item in enumerate(sequence):
        if item != sequence[-(i+1)]:
            return False
    return True
```

Также следует обратить внимание, что новички в Python часто используют range(len(sequence)). Если вы когда-нибудь встретите конструкции типа range(len(...)), лучше заменить её на enumerate. Она поможет упростить конструкцию операторов.

```
def palindromic(sequence):
     """Возвращает True, если последовательность является палиндромом."""
    for i in range(len(sequence)):
        if sequence[i] != sequence[-(i+1)]:
            return False
    return True
```

zip

Эта функция ещё более специализирована, чем enumerate. Zip используется для перебора сразу нескольких объектов одновременно.

```
>>> one_iterable = [2, 1, 3, 4, 7, 11]
```

```
>>> another_iterable = ['P', 'y', 't', 'h', 'o', 'n']
>>> for n, letter in zip(one_iterable, another_iterable):
...     print(letter, n)
...
P 2
y 1
t 3
h 4
o 7
n 11
```

По сравнению с enumerate, последняя функция удобна, когда нужна индексация во время цикла. Если нужно обрабатывать несколько объектов одновременно, zip предпочтительнее enumerate.

reversed

Функция reversed, как enumerate и zip, возвращает итератор.

```
>>> numbers = [2, 1, 3, 4, 7]
>>> reversed(numbers)
```

Единственное, что можно сделать с этим итератором, пройтись по нему (но только один раз).

```
>>> reversed_numbers = reversed(numbers)
>>> list(reversed_numbers)
[7, 4, 3, 1, 2]
>>> list(reversed_numbers)
[]
```

Подобно enumerate и zip, reversed является своего рода вспомогательной функцией в циклах. Её использование можно увидеть исключительно в цикле for.

```
>>> for n in reversed(numbers):
...     print(n)
```

```
...
7
4
3
1
2
```

Есть несколько и других способов перевернуть списки в Python.

```
# Синтаксис нарезки
for n in numbers[::-1]:
    print(n)

# Метод переворота на месте
numbers.reverse()
for n in numbers:
    print(n)
```

Данная функция, как правило, является лучшим способом «перевернуть» любой список (а также набор, массив и т. д.) в Python. В отличие от numbers.reverse(), reversed не изменяет список, а возвращает итератор перевёрнутых элементов. reversed(numbers) (в отличие от numbers [::-1]) не создает новый список. Возвращаемый им итератор извлекает следующий элемент в обратном порядке при проходе по циклу. Также синтаксис reversed(numbers) намного более читабелен, чем numbers [::-1].

Можно использовать факт, что reversed не копирует список. Если объединить его с функцией zip, можно переписать функцию palindromic (из раздела enumerate), не занимая дополнительной памяти (не копируя объект).

```
def palindromic(sequence):
     """Возвращает True, если последовательность является палиндромом."""
    for n, m in zip(sequence, reversed(sequence)):
        if n != m:
            return False
```

```
    return True
```

sum

Эта функция берёт набор чисел и возвращает их сумму.

```
>>> sum([2, 1, 3, 4, 7])
17
```

В Python есть много вспомогательных функций, которые выполняют циклы за вас (отчасти потому, что они хорошо сочетаются с генератор-выражениями).

```
>>> numbers = [2, 1, 3, 4, 7, 11, 18]
>>> sum(n**2 for n in numbers)
524
```

min и max

Эти функции выдают минимальное и максимальное число из набора соответственно.

```
>>> numbers = [2, 1, 3, 4, 7, 11, 18]
>>> min(numbers)
1
>>> max(numbers)
18
```

Данные методы сравнивают элементы, используя оператор <. Поэтому, все передаваемые в них значения должны быть упорядочены и сопоставимы друг с другом. Min и max также принимают key-свойство, позволяющее настроить, что на самом деле означают «минимум» и «максимум» для конкретных объектов.

```
>>> fruits = ['kumquat', 'Cherimoya', 'Loquat', 'longan',
'jujube']
>>> sorted(fruits, key=len)
['Loquat', 'longan', 'jujube', 'kumquat', 'Cherimoya']
```

sorted

Эта функция принимает любой набор элементов и возвращает новый список всех значений в отсортированном порядке.

```
>>> numbers = [1, 8, 2, 13, 5, 3, 1]
>>> words = ["python", "is", "lovely"]
>>> sorted(words)
['is', 'lovely', 'python']
>>> sorted(numbers, reverse=True)
[13, 8, 5, 3, 2, 1, 1]
```

Данная функция (как min и max) сравнивает элементы, используя оператор <, поэтому все значения, переданные ей, должны быть упорядочены. Sorted также позволяет настраивать сортировку с помощью key-свойства.

any и all

Эти функции могут быть использованы в паре с генератор-выражениями, чтобы определить соответствие элементов заданному условию.

Используя all, можно переписать функцию palindromic следующим образом.

```
def palindromic(sequence):
     """Возвращает True, если последовательность является палиндромом."""
    return all(
        n == m
        for n, m in zip(sequence, reversed(sequence))
    )
```

Отрицание условия и возвращаемого значения позволит также использовать any в этом примере точно также (что усложнит конструкцию, но вполне сойдёт в качестве примера использования).

```
def palindromic(sequence):
     """Возвращает True, если последовательность является палиндромом."""
    return not any(
        n != m
        for n, m in zip(sequence, reversed(sequence))
    )
```

5 функций для отладки

Эти функции часто игнорируются, но будут полезны для отладки и устранения неисправностей кода.

breakpoint

Если нужно приостановить выполнение кода и перейти в командную строку Python, эта функция вам пригодится. Вызов breakpoint перебросит вас в отладчик Python. Эта встроенная функция была добавлена в Python 3.7, но если вы работаете в более старых версиях, можете получить тот же результат с помощью import pdb; pdb.set_trace().

dir

Эта функция может использоваться в двух случаях:

- просмотр списка всех локальных переменных;
- просмотр списка всех атрибутов конкретного объекта.

Из примера можно увидеть локальные переменные сразу после запуска и после создания новой переменной x.

```
>>> dir()
['__annotations__', '__doc__', '__name__', '__package__']
>>> x = [1, 2, 3, 4]
>>> dir()
['__annotations__', '__doc__', '__name__', '__package__', 'x']
```

Если в dir передать созданный список x, на выходе можно

увидеть все его атрибуты.

```
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

В выведенном списке атрибутов можно увидеть его типичные методы (append, pop, remove и т. д.) , а также множество более сложных методов для перегрузки операторов.

vars

Эта функция является своего рода смесью двух похожих инструментов: locals() и __dict__. Когда vars вызывается без аргументов, это эквивалентно вызову locals(), которая показывает словарь всех локальных переменных и их значений.

```
>>> vars()
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': , '__spec__': None, '__annotations__': {},
'__builtins__': }
```

Когда вызов происходит с аргументом, vars получает доступ к атрибуту __dict__, который представляет собой словарь всех атрибутов экземпляра.

```
>>> from itertools import chain
>>> vars(chain)
mappingproxy({'__getattribute__': , '__iter__': , '__next__':
```

, '__new__': , 'from_iterable': , '__reduce__': , '__setstate__': , '__doc__': 'chain(*iterables) --> chain object\n\nReturn a chain object whose .__next__() method returns elements from the\nfirst iterable until it is exhausted, then elements from the next\niterable, until all of the iterables are exhausted.'})

Перед использованием vars было бы неплохо сначала обратиться к dir.

type

Эта функция возвращает тип объекта, который вы ей передаете.

Тип экземпляра класса есть сам класс.

```
>>> x = [1, 2, 3]
>>> type(x)
```

Тип класса — это его метакласс, обычно это type.

```
>>> type(list)

>>> type(type(x))
```

Атрибут __class__ даёт тот же результат, что и функция type, но рекомендуется использовать второй вариант.

```
>>> x.__class__
>>> type(x)
```

Функция type, кроме отладки, иногда полезна и в реальном коде (особенно в объектно-ориентированном программировании с наследованием и пользовательскими строковыми представлениями). Обратите внимание, что при проверке типов обычно вместо type используется функция isinstance. Также стоит понимать, что в Python обычно не принято проверять типы объектов (вместо этого практикуется утиная типизация).

help

Если вы находитесь в Python Shell или делаете отладку кода с использованием breakpoint, и хотите знать, как работает определённый объект, метод или атрибут, функция help поможет вам. В действительности вы, скорее всего, будете обращаться за помощью к поисковой системе. Но если вы уже находитесь в Python Shell, вызов help(list.insert) будет быстрее, чем поиск документации в Google.

Функции, которые пригодятся позже

В начале изучения Python эти функции вам по большей части будут не нужны, но в конечном итоге они вам понадобятся.

open

Эта функция служит для открытия файла и последующей работы с ним. Но, если вы не работаете с файлами напрямую, то она вам может и не пригодиться. Несмотря на то, что работа с файлами очень распространена, далеко не все программисты Python работают с файлами через open. Например, разработчики Django могут не использовать её вообще.

input

Эта функция запрашивает у пользователя ввод, ждёт нажатия клавиши Enter, а затем возвращает набранный текст. Чтение из стандартного ввода — это один из способов получить входные данные в программе. Но есть и много других способов: аргументы командной строки, чтение из файла, чтение из базы данных и многое другое.

repr

Эта функция необходима для представления объекта в читабельном виде. Для многих объектов функции str и repr работают одинаково.

```
>>> str(4), repr(4)
('4', '4')
>>> str([]), repr([])
('[]', '[]')
```

Но есть объекты, для которых их применение различается.

```
>>> str('hello'), repr("hello")
('hello', "'hello'")
>>> from datetime import date
>>> str(date(2020, 1, 1)), repr(date(2020, 1, 1))
('2020-01-01', 'datetime.date(2020, 1, 1)')
```

Строковое представление, которое вы видите в Python Shell, использует repr, тогда как функция print использует str.

```
>>> date(2020, 1, 1)
datetime.date(2020, 1, 1)
>>> "hello!"
'hello!'
>>> print(date(2020, 1, 1))
2020-01-01
>>> print("hello!")
hello!
```

Также repr используется при ведении лог-журнала, обработке исключений и реализации более сложных методов.

super

Эта функция очень важна, если используется наследование одного класса от другого. Многие пользователи Python редко создают классы. Они не являются важной частью Python, хоть для многих типов программирования они необходимы. Например, вы не можете использовать веб-фреймворк Django без создания классов.

property

Эта функция является декоратором и дескриптором.

Декоратор позволяет создать атрибут, который всегда будет содержать возвращаемое значение конкретного вызова функции. Это проще всего понять на примере. Пример класса, который использует property.

```
class Circle:

    def __init__(self, radius=1):
        self.radius = radius

    @property
    def diameter(self):
        return self.radius * 2
```

Здесь вы можете увидеть доступ к атрибуту diameter объекта Circle.

```
>>> circle = Circle()
>>> circle.diameter
2
>>> circle.radius = 5
>>> circle.diameter
10
```

Если вы занимаетесь объектно-ориентированным программированием на Python, вам, вероятно, захочется узнать о property больше в какой-то момент. В отличие от других объектно-ориентированных языков, в Python property используется вместо методов getter и setter.

issubclass и isinstance

Функция issubclass проверяет, является ли класс подклассом одного или нескольких других классов.

```
>>> issubclass(int, bool)
False
>>> issubclass(bool, int)
```

```
True
>>> issubclass(bool, object)
True
```

Функция isinstance проверяет, является ли объект экземпляром одного или нескольких классов.

```
>>> isinstance(True, str)
False
>>> isinstance(True, bool)
True
>>> isinstance(True, int)
True
>>> isinstance(True, object)
True
```

Функция isinstance может быть представлена как делегирование в issubclass.

```
>>> issubclass(type(True), str)
False
>>> issubclass(type(True), bool)
True
>>> issubclass(type(True), int)
True
>>> issubclass(type(True), object)
True
```

Если вы перегружаете операторы, вам может понадобиться использование isinstance, но в целом в Python стараются избегать строгой проверки типов, поэтому особой нужды в данных функциях нет.

hasattr, getattr, setattr и delattr

Если нужно работать с атрибутами объекта, но имя атрибутов является динамическим и постоянно меняется, данные функции вам будут необходимы. Например, есть объект thing, который нужно проверить на конкретное значение.

```
>>> class Thing: pass
...
>>> thing = Thing()
```

Функция hasattr позволяет проверить, имеет ли объект определённый атрибут.

```
>>> hasattr(thing, 'x')
False
>>> thing.x = 4
>>> hasattr(thing, 'x')
True
```

Функция getattr позволяет получить значение атрибута (с необязательным значением по умолчанию, если атрибут не существует).

```
>>> getattr(thing, 'x')
4
>>> getattr(thing, 'x', 0)
4
>>> getattr(thing, 'y', 0)
0
```

Функция setattr позволяет установить значение атрибута.

```
>>> setattr(thing, 'x', 5)
>>> thing.x
5
```

И delattr соответственно удаляет атрибут.

```
>>> delattr(thing, 'x')
>>> thing.x
Traceback (most recent call last):
  File "", line 1, in
AttributeError: 'Thing' object has no attribute 'x'>
```

classmethod и staticmethod

Если у вас есть метод, который должен вызываться в экземпляре или в классе, вам нужен декоратор classmethod. Фабричные методы (альтернативные конструкторы) являются распространённым случаем для этого.

```
class RomanNumeral:

    """Римская цифра, представленная как строка и число."""

    def __init__(self, number):
        self.value = number

    @classmethod
    def from_string(cls, string):
         return cls(roman_to_int(string))  # Функция пока еще не существует
```

Немного сложнее придумать хорошее использование staticmethod, так как всегда можно использовать функцию уровня модуля вместо статического метода.

```
class RomanNumeral:

    """Римская цифра, представленная как строка и число."""

    SYMBOLS = {'M': 1000, 'D': 500, 'C': 100, 'L': 50, 'X': 10, 'V': 5, 'I': 1}

    def __init__(self, number):
        self.value = number

    @classmethod
    def from_string(cls, string):
        return cls(cls.roman_to_int(string))

    @staticmethod
    def roman_to_int(numeral):
```

```
        total = 0
          for symbol, next_symbol in zip_longest(numeral,
numeral[1:]):
            value = RomanNumeral.SYMBOLS[symbol]
            next_value = RomanNumeral.SYMBOLS.get(next_symbol,
0)
            if value < next_value:
                value = -value
            total += value
        return total
```

Функция roman_to_int не требует доступа к экземпляру или классу, поэтому ей не нужно быть @classmethod. Фактически нет необходимости делать эту функцию staticmethod (вместо classmethod). staticmethod является просто более сдерживающим, чтобы сигнализировать о том факте, что функция не зависима от класса, в котором она находится.

next

Данная функция возвращает следующий элемент в итераторе. Она может работать со следующими видами итераторов:

- объекты enumerate;
- объекты zip;
- возвращаемые значения функции reversed;
- файлы
- объекты csv.reader;
- генератор-выражения;
- генератор-функции;

Функция next может быть представлена как способ вручную перебрать набор, чтобы получить один единственный элемент, а затем выйти из перебора.

Функции, которые когда-нибудь можно

выучить

Следующие встроенные функции Python определённо не бесполезны, но они более специализированы. Эти функции вам, возможно, будут нужны, но также есть шанс, что вы никогда не прибегнете к ним в своём коде.

- iter: возвращает итератор (список, набор и т. д.);
- callable: возвращает True, если аргумент является вызываемым;
- filter and map: вместо них рекомендуется использовать генератор-выражения;
- Round: округляет число;
- divmod: эта функция выполняет деление без остатка (//) и операцию по модулю (%) одновременно;
- bin, oct и hex: служат для отображения чисел в виде строки в двоичной, восьмеричной или шестнадцатеричной форме;
- abs: возвращает абсолютное значение числа (аргумент может быть целым или числом с плавающей запятой, если аргумент является комплексным числом, его величина возвращается);
- hash;
- object.

Прочие специфические функции

- ord и chr: могут пригодиться при изучении ASCII или Unicode;
- exec и eval: для исполнения строки;
- compile;
- slice: если вы реализуете __getitem__ для создания пользовательской последовательности, это может вам понадобиться;
- bytes, bytearray и memoryview: если вы часто работаете с байтами;
- ascii: похож на repr, но возвращает представление

объекта только в ASCII;
- frozenset: как set, но он неизменен (и хешируемый);
- __import__: лучше использовать importlib;
- format: вызывает метод __format__, который используется для форматирования строк;
- pow: оператор возведения в степень (**);
- complex: если вы не используете комплексные числа (4j + 3), она вам не понадобится.

Заключение

Если вы только начинаете свой путь в изучении Python, нет необходимости изучать все встроенные функции сейчас. Не торопитесь, сосредоточьтесь на первых двух пунктах (общеизвестные и упускаемые из виду), а □□после можете перейти и к другим, если/когда они вам понадобятся.

Использованы материалы: tproger.ru