Intro

Комментирование

Комментарии - это фрагменты текста, которые находятся в коде, но игнорируются интерпретатором Python при выполнении кода. Можно использовать комментарии для описания кода, чтобы вы и другие разработчики могли быстро понять, что делает код или почему код написан определенным образом. Чтобы написать комментарий на Python, просто добавьте решетку (#) перед текстом комментария:

```
# Это комментарий в отдельной строке
```

Интерпретатор Python игнорирует текст после решетки и до конца строки. Вы также можете добавить в свой код встроенные комментарии. Другими словами, вы можете объединить выражение или оператор Python с комментарием в одной строке, учитывая, что комментарий занимает последнюю часть строки:

```
var = "Привет, мир!" # Это встроенный комментарий
```

Комментарии должны быть краткими и по существу. PEP 8 рекомендует оставлять комментарии длиной не более 72 символов. Если ваш комментарий приближается к этой длине или превышает ее, вы можете разместить его на нескольких строках:

```
# Это длинный комментарий, требующий
# две строки для завершения.
```

Если вам нужно больше места для данного комментария, вы можете использовать несколько строк с решеткой на каждой.

Строки документации (**docstrings**) служат той же цели, что и обычные комментарии, поскольку они предназначены для объяснения кода. Однако они более конкретны и имеют другой синтаксис. Они создаются путем помещения многострочной строки, содержащей объяснение функции, под ее первой строкой:

```
def shout(word):
    """
    Печатает слово с
    восклицательным знаком в конце
    """
    print(word + "!")

shout("spam")
```

```
spam!
```

Переменные

В Питоне переменные - это имена, прикрепленные к определенному объекту. Они содержат ссылку или указатель на адрес памяти, по которому хранится объект. После того, как переменной назначен объект, вы можете получить доступ к объекту, используя имя переменной. Вам необходимо заранее определить свои переменные. Вот синтаксис:

```
variable_name = variable_value
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-e916d0a9b3b1> in <module>
----> 1 variable_name = variable_value

NameError: name 'variable_value' is not defined
```

Вы должны использовать схему именования, которая сделает ваши переменные интуитивно понятными и удобно читаемыми. Имя переменной должно указывать на присвоенные ей значения.

Иногда программисты используют короткие имена переменных, например x и y. Это совершенно подходящие имена в контексте математики, алгебры и так далее. В других контекстах вам следует избегать одно символьных имен и использовать что-то более наглядное. Таким образом, другие разработчики могут сделать обоснованное предположение о том, что содержат ваши переменные. При написании программ думайте о других, а также о себе в будущем.

Вот несколько примеров допустимых и недопустимых имен переменных в Python:

```
numbers = [1, 2, 3, 4, 5]
numbers
```

```
[1, 2, 3, 4, 5]
```

```
first_num = 1
first_num
```

```
1
```

```
1rst_num = 1
```

```
  File "<ipython-input-8-20ce1ddde2db>", line 1
    1rst_num = 1
     ^
SyntaxError: invalid syntax
```

Имена переменных могут быть любой длины и состоять из прописных и строчных букв (A-Z, a-z), цифр (0-9), а также символа подчеркивания (_). В общем, имена переменных должны быть буквенно-цифровыми, но учтите, что хотя имена переменных могут содержать цифры, их первый символ не может быть цифрой.

> **Примечание**
>
> Соглашение об именах lower_case_with_underscores, также известное как **snake_case**, обычно используется в Python.

Ключевые слова

Как и в любом другом языке программирования, в Python есть набор специальных слов, которые являются частью его синтаксиса. Эти слова известны как ключевые слова. Чтобы получить полный список ключевых слов, доступных в вашей текущей установке Python, вы можете запустить следующий код в консоли:

```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Каждое из этих ключевых слов играет роль в синтаксисе Python. Это зарезервированные слова, которые имеют определенные значения и цели в языке, поэтому вы не должны использовать их ни для чего, кроме этих конкретных целей. Например, вы не должны использовать их в качестве имен переменных в своем коде.

Есть еще один способ получить доступ ко всему списку ключевых слов Python:

```
import keyword
keyword.kwlist
```

```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

keyword предоставляет набор функций, которые позволяют вам определить, является ли данная строка ключевым словом. Например, keyword.kwlist содержит список всех текущих ключевых слов в Python. Это удобно, когда вам нужно программно манипулировать ключевыми словами в ваших программах Python.

Ввод и вывод в консоль

Чтение ввода с клавиатуры

Программам часто требуется получать данные от пользователя, обычно путем ввода с клавиатуры. Самый простой способ сделать это в Python - использовать `input()`.

```
input([<prompt>])  # Читает строку ввода с клавиатуры.
```

`input()` приостанавливает выполнение программы, чтобы пользователь мог ввести строку с клавиатуры. Как только пользователь нажимает клавишу Enter, все набранные символы считываются и возвращаются в виде строки:

```
s = input()
s
```

```
word
```

```
'word'
```

Обратите внимание, что новая строка, генерируемая, когда пользователь нажимает клавишу Enter, не включается в возвращаемую строку.

Если вы включите необязательный аргумент `<prompt>`, `input()` отобразит его как приглашение пользователю перед приостановкой чтения ввода:

```
name = input('What is your name? ')
name
```

`input()` всегда возвращает строку. Если вам нужен числовой тип, вам нужно преобразовать строку в соответствующий тип с помощью встроенных функций `int()`, `float()` или `complex()`:

```
n = input('Enter a number: ')
n + 100
```

```
Enter a number: 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-35b5a12a8d74> in <module>
      1 n = input('Enter a number: ')
----> 2 n + 100

TypeError: can only concatenate str (not "int") to str
```

```
n = int(input('Enter a number: '))
n + 100
```

```
Enter a number: 10
```

```
110
```

В приведенном выше примере выражение n + 100 в строке 2 недопустимо, потому что n - строка, а 100 - целое число. Строка 3 преобразует n в целое число, поэтому операция в строке 4 завершается успешно.

Запись вывода в консоль

Помимо получения данных от пользователя, программе также обычно необходимо предоставлять данные обратно пользователю. Можно отображать данные в консоли в Python с помощью `print()`.

Чтобы вывести объекты в консоли, передайте их как список аргументов, разделенных запятыми, в `print()`.

```
print (<obj>, ..., <obj>)  # Отображает строковое представление каждого <obj> в консоли.
```

По умолчанию `print()` разделяет каждый объект одним пробелом и добавляет новую строку в конец вывода:

```
fname = 'Winston'
lname = 'Smith'
print('Name:', fname, lname)
```

```
Name: Winston Smith
```

В качестве аргумента функции `print()` можно указать любой тип объекта. Если объект не является строкой, то `print()` преобразует его в соответствующее строковое представление, отображающее его:

```
a = [1, 2, 3]
type(a)

b = -12
type(b)

d = {'foo': 1, 'bar': 2}
type(d)

type(len)

print(a, b, d, len)
```

```
[1, 2, 3] -12 {'foo': 1, 'bar': 2} <built-in function len>
```

Как видите, даже сложные типы, такие как списки, словари и функции, могут отображаться на консоли с помощью `print()`.

Аргументы print()

`print()` принимает несколько дополнительных аргументов, которые обеспечивают контроль над форматом вывода. Каждый из них представляет собой особый тип аргумента, называемый аргументом ключевого слова.

- Аргументы ключевого слова имеют вид `<keyword>=<value>`.
- Любые аргументы ключевого слова, передаваемые в `print()`, должны идти в конце после списка отображаемых объектов.

Аргумент sep=

Добавление аргумента `sep = <str>` приводит к тому, что объекты разделяются строкой `<str>` вместо одиночного пробела по умолчанию:

```
print('foo', 42, 'bar')
```

```
foo 42 bar
```

```
print('foo', 42, 'bar', sep='/')
```

```
foo/42/bar
```

```
print('foo', 42, 'bar', sep='...')
```

```
foo...42...bar
```

```
d = {'foo': 1, 'bar': 2, 'baz': 3}
for k, v in d.items():
    print(k, v, sep=' -> ')
```

```
foo -> 1
bar -> 2
baz -> 3
```

Чтобы сжимать объекты вместе без пробелов между ними, укажите `sep = ''`:

```
print('foo', 42, 'bar', sep='')
```

```
foo42bar
```

Аргумент end=

Аргумент `end = <str>` заставляет вывод завершаться `<str>` вместо новой строки по умолчанию:

```
print('foo', end='/')
print(42, end='/')
print('bar')
```

```
foo/42/bar
```

Data Types

Встроенные типы данных

Python имеет несколько встроенных типов данных, таких как **числа** (целые числа, числа с плавающей запятой, комплексные числа), **логические значения**, **строки**, **списки**, **кортежи**, **словари** и **множества**. Ими можно управлять с помощью нескольких инструментов:

- Операторы
- Встроенные функции
- Методы типа данных

Числа

Python предоставляет **целые числа**, **числа с плавающей запятой** и **комплексные числа**. Целые числа и числа с плавающей запятой являются наиболее часто используемыми числовыми типами в повседневном программировании, в то время как комплексные числа имеют особые варианты использования в математике и естественных науках.

Вот краткое описание их функций:

Number	Description	Examples	Python Data Type
Integer	Целые числа	1, 2, 42, 476, -99999	int
Floating-point	Числа с плавающей запятой	1.0, 2.2, 42.09, 476.1, -99999.9	float
Complex	Числа с действительной и мнимой частью	complex(1, 2), complex(-1, 7), complex("1+2j")	complex

Целые числа имеют неограниченную точность. Информация о точности чисел с плавающей запятой доступна в `sys.float_info`. Комплексные числа имеют действительную и мнимую части, которые являются числами с плавающей запятой.

Логические значения

Логические значения реализованы как подкласс целых чисел с двумя возможными значениями в Python: **True** или **False**. Обратите внимание, что эти значения должны начинаться с заглавной буквы.

Строки

Строки - это фрагменты текста или последовательности символов, которые можно определить с помощью одинарных, двойных или тройных кавычек:

```
# Использование одинарных кавычек
greeting = 'Hello there!'

# Использование двойных кавычек
welcome = "Hello there!"

# Использование тройных кавычек
message = """Hello there!"""
```

```
# Экранирование символов
escaped = 'can\'t'
print('Escaped: ', escaped)

not_escaped = "can't"
print('Not escaped: ', not_escaped)
```

```
Escaped:  can't
Not escaped:  can't
```

Обратите внимание, что вы можете использовать разные типы кавычек для создания строковых объектов в Python. Вы также можете использовать символ обратной косой черты () для экранирования символов со специальным значением, таких как сами кавычки.

Списки

Списки обычно называются **массивами** почти во всех других языках программирования. В Python списки - это изменяемые последовательности, которые группируют различные объекты вместе. Чтобы создать список, вы используете присваивание с последовательностью объектов, разделенных запятыми, в квадратных скобках ([]) с правой стороны:

```
# Определение пустого списка
empty = []

# Определение списка чисел
numbers = [1, 2, 3, 100]

# Изменение списка
numbers[3] = 200

# Определение списка строк
superheroes = ["batman", "superman", "spiderman"]

# Определение списка объектов с разными типами данных
mixed_types = ["Hello World", [4, 5, 6], False]
```

Списки могут содержать объекты разных типов данных, в том числе другие списки. Они также могут быть пустыми. Поскольку списки являются изменяемыми последовательностями, вы можете изменять их на месте, используя индексную нотацию и операцию присваивания.

Кортежи

Кортежи похожи на списки, но представляют собой неизменяемые последовательности. Это означает, что вы не можете изменить их после создания. Чтобы создать объект кортежа, можно использовать операцию присваивания с последовательностью элементов, разделенных запятыми, с правой стороны. Обычно для разделения кортежа используются круглые скобки, но они не являются обязательными:

```
employee = ("Jane", "Doe", 31, "Software Developer")

employee[0] = "John"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-b12d1c27d9d5> in <module>
      1 employee = ("Jane", "Doe", 31, "Software Developer")
      2
----> 3 employee[0] = "John"

TypeError: 'tuple' object does not support item assignment
```

Если вы попытаетесь изменить кортеж на месте, вы получите ошибку TypeError, указывающую на то, что кортежи не поддерживают модификации.

Словари

Словари - это тип ассоциативного массива, содержащего набор пар ключ-значение, в котором каждый ключ является хешируемым объектом, который сопоставляется с произвольным объектом, значением. Есть несколько способов создать словарь. Вот два из них:

```
person1 = {"name": "John Doe", "age": 25, "job": "Python Developer"}
person2 = dict(name="Jane Doe", age=24, job="Web Developer")

print(person1)
print(person2)
```

```
{'name': 'John Doe', 'age': 25, 'job': 'Python Developer'}
{'name': 'Jane Doe', 'age': 24, 'job': 'Web Developer'}
```

Первый подход использует пару фигурных скобок, в которые вы добавляете разделенный запятыми список пар ключ-значение, используя двоеточие (:) для отделения ключей от значений. Второй подход использует встроенную функцию `dict()`, которая может принимать аргументы ключевых слов и превращать их в словарь с ключевыми словами в качестве ключей и аргументами в качестве значений.

Множества

Python также предоставляет такую структуру данных, как множество. Множества - это неупорядоченные и изменяемые коллекции произвольных, но хешируемых объектов Python. Создавать множества можно несколькими способами. Вот два из них:

```
employees1 = {"John", "Jane", "Linda"}

employees2 = set(["David", "Mark", "Marie"])

empty = set()
```

В первом примере для создания множества используются фигурные скобки и список объектов, разделенных запятыми. Если вы используете `set()`, вам необходимо предоставить итерацию с объектами, которые вы хотите включить в набор. Наконец, если вы хотите создать пустой набор, вам нужно использовать `set()` без аргументов. Использование пустой пары фигурных скобок создает пустой словарь вместо множества.

Один из наиболее распространенных вариантов использования множеств - их использование для удаления повторяющихся объектов из существующего итеративного объекта:

```
set ([1, 2, 2, 3, 4, 5, 3])
```

```
{1, 2, 3, 4, 5}
```

Поскольку наборы являются коллекциями уникальных объектов, когда вы создаете набор с помощью `set()` и итерируемого объекта в качестве аргумента, конструктор класса удаляет все повторяющиеся объекты и сохраняет только по одному экземпляру каждого из них в результирующем наборе.

Когда использовать словарь:

- Когда вам нужна логическая связь между парой key:value.
- Когда вам нужен быстрый поиск ваших данных на основе настраиваемого ключа.
- Когда ваши данные постоянно модифицируются. Помните, словари изменчивы.

Когда использовать другие типы:

- Используйте списки, если у вас есть набор данных, не требующий произвольного доступа. Старайтесь выбирать списки, когда вам нужна простая повторяющаяся коллекция, которая часто изменяется.
- Используйте множество, если вам нужна уникальность элементов.
- Используйте кортежи, когда ваши данные не могут измениться.

Операции над числами

Операторы представляют такие операции, как сложение, вычитание, умножение, деление и т. д. Когда вы комбинируете их с числами, они образуют выражения, которые Python может вычислить:

```
# Сложение
5 + 3

# Вычитание
5 - 3

# Умножение
5 * 3

# Деление
5 / 3

# Целочисленное деление
5 // 3

# Остаток от деления
5 % 3

# Возведение в степень
5 ** 3
```

Эти операторы работают с двумя **операндами** и обычно называются **арифметическими операторами**. Операнды могут быть числами или переменными, которые содержат числа.

Передавая целое число или строку, представляющую число, `float()` возвращает число с плавающей запятой:

```
# Целые числа
float(9)
```

```
9.0
```

```
# Строки, представляющие числа
print(float("2"))
print(float("-200"))
print(float("2.25"))
```

```
2.0
-200.0
2.25
```

```
# Комплексные числа
float(complex(1, 2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-e88d092651d1> in <module>
      1 # Комплексные числа
----> 2 float(complex(1, 2))

TypeError: can't convert complex to float
```

С помощью `float()` вы можете преобразовывать целые числа и строки, представляющие числа, в числа с плавающей запятой, но вы не можете преобразовывать комплексное число в число с плавающей запятой.

Учитывая число с плавающей запятой или строку, `int()` возвращает целое число. Эта функция не округляет до ближайшего целого числа. Она просто обрезает, отбрасывая все, что находится после десятичной точки, и возвращает число. Итак, ввод 10,6 возвращает 10 вместо 11. Аналогично, 3,25 возвращает 3:

```
# Числа с плавающей запятой
int(10.6)
```

```
10
```

```
# Строки, представляющие числа
print(int("2"))
print(int("2.3"))
```

```
2
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-82cbdc68d355> in <module>
      1 # Строки, представляющие числа
      2 print(int("2"))
----> 3 print(int("2.3"))

ValueError: invalid literal for int() with base 10: '2.3'
```

```
# Комплексные числа
int(complex(1, 2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-b45c6433f10d> in <module>
      1 # Комплексные числа
----> 2 int(complex(1, 2))

TypeError: can't convert complex to int
```

Обратите внимание, что вы можете передать строку, представляющую целое число, в `int()`, но вы не можете передать строку, представляющую число с плавающей запятой. Комплексные числа тоже не работают.

Помимо этих встроенных функций, с каждым типом числа связано несколько методов. Вы можете получить к ним доступ, используя ссылку на атрибут, также известную как точечная нотация (**dot notation**):

```
10.0.is_integer()
```

```
True
```

```
10.2.is_integer()
```

```
False
```

```
(10).bit_length()
```

```
4
```

```
10.bit_length()
```

```
  File "<ipython-input-22-2c75d808345c>", line 1
    10.bit_length()
       ^
SyntaxError: invalid syntax
```

Эти методы могут быть полезным инструментом для изучения. В случае целых чисел для доступа к их методам через литерал (**literal**) необходимо использовать пару круглых скобок. В противном случае вы получите SyntaxError.

Логические выражения и операторы

Логические значения используются, чтобы выразить значение истинности выражения или объекта. Логические значения удобны при написании функций-предикатов или при использовании операторов сравнения, таких как больше (>), меньше (<), равно (==) ...

```
2 < 5
```

```
True
```

```
4 > 10
```

```
False
```

```
4 <= 3
```

```
False
```

```
3 >= 3
```

```
True
```

```
5 == 6
```

```
False
```

```
6 != 9
```

```
True
```

Python предоставляет встроенную функцию `bool()`, которая тесно связана с логическими значениями. Вот как это работает:

```
bool(0)
```

```
False
```

```
bool(1)
```

```
True
```

```
bool("")
```

```
False
```

```
bool("a")
```

```
True
```

```
bool([])
```

```
False
```

```
bool([1, 2, 3])
```

```
True
```

`bool()` принимает объект в качестве аргумента и возвращает True или False в соответствии с истинным значением объекта.

`int()` принимает логическое значение и возвращает 0 для False и 1 для True:

```
int(False)
```

```
0
```

```
int(True)
```

```
1
```

Это связано с тем, что Python реализует свои логические значения как подкласс `int`.

Операции над строками

После определения строк можно использовать оператор «плюс» (+), чтобы объединить их в новую строку:

```
"Happy" + " " + "pythoning!"
```

```
'Happy pythoning!'
```

При использовании со строками оператор «плюс» (+) объединяет их в одну строку. Обратите внимание, что вам нужно включить пробел (" ") между словами, чтобы иметь правильный интервал в полученной строке. Если нужно объединить много строк, вам следует подумать об использовании `.join()`, что более эффективно. Вы узнаете о `.join()` немного позже.

Python имеет множество полезных встроенных функций и методов для работы со строками. Например, если вы передадите строку в качестве аргумента функции `len()`, вы получите длину строки или количество содержащихся в ней символов:

```
len("Happy pythoning!")
```

```
16
```

Строковый класс (`str`) предоставляет богатый набор методов, полезных для манипулирования и обработки строк. `.join()` берет итерацию строк и объединяет их в новую строку. Строка, для которой вы вызываете метод, играет роль разделителя:

```
" ".join(["Happy", "pythoning!"])
```

```
'Happy pythoning!'
```

`.upper()` возвращает копию базовой строки со всеми буквами, преобразованными в верхний регистр:

```
"Happy pythoning!".upper()
```

```
'HAPPY PYTHONING!'
```

`.lower()` возвращает копию базовой строки со всеми буквами, преобразованными в нижний регистр:

```
"HAPPY PYTHONING!".lower()
```

```
'happy pythoning!'
```

`.format()` выполняет операцию форматирования строки. Этот метод обеспечивает большую гибкость при форматировании и интерполяции строк:

```
name = "John Doe"
age = 25
"My name is {0} and I'm {1} years old".format(name, age)
```

```
"My name is John Doe and I'm 25 years old"
```

Вы также можете использовать **f-строку** для форматирования ваших строк без использования `.format()`:

```
name = "John Doe"
age = 25
f"My name is {name} and I'm {age} years old"
```

```
"My name is John Doe and I'm 25 years old"
```

F-строки - это улучшенный синтаксис форматирования строк. Это строковые литералы с буквой f в начале вне кавычек. Выражения, заключенные в фигурные скобки ({}), заменяются своими значениями в форматированной строке.

Строки - это последовательности символов. Это означает, что вы можете извлекать отдельные символы из строки, используя их позиционный индекс. Индекс - это целое число с отсчетом от нуля, связанное с определенной позицией в последовательности:

```
welcome = "Welcome to Python!"
print(
    welcome[0],
    welcome[11],
    welcome[-1],
    sep='\n'
)
```

```
W
P
!
```

Операция индексации извлекает символ в позиции, указанной данным индексом. Обратите внимание, что отрицательный индекс извлекает элемент в обратном порядке, где -1 является индексом последнего символа в строке.

Вы также можете получить часть строки, разрезав ее:

```
welcome = "Welcome to Python!"
print(
    welcome[0:7],
    welcome[11:17],
    sep='\n'
)
```

```
Welcome
Python
```

Операция **slice** принимает элемент в форме `[start:end:step]`. Здесь start - это индекс первого элемента, который нужно включить в срез, а end - это индекс последнего элемента, который не включен в возвращаемый срез. Наконец, step - необязательное целое число, представляющее количество элементов, через которые нужно перейти при извлечении элементов из исходной строки. Например, шаг 2 вернет каждый второй элемент между start и end.

Операции над списками

Списки представляют собой последовательности, подобные строкам, поэтому можно получить доступ к их отдельным элементам, используя целочисленные индексы с отсчетом от нуля:

```
numbers = [1, 2, 3, 200]
print(
    numbers[0],
    numbers[1],
    sep='\n'
)
```

```
1
2
```

```
superheroes = ["batman", "superman", "spiderman"]
print(
    superheroes[-1],
    superheroes[-2],
    sep='\n'
)
```

```
spiderman
superman
```

Вы также можете создавать новые списки из существующего списка, используя операцию `slice`:

```
numbers = [1, 2, 3, 200]
new_list = numbers[0:3]
new_list
```

```
[1, 2, 3]
```

Если вы вложите список, строку или любую другую последовательность в другой список, вы можете получить доступ к внутренним элементам, используя несколько индексов:

```
mixed_types = ["Hello World", [4, 5, 6], False]
mixed_types[1][2]
```

```
6
```

Вы также можете объединить свои списки с помощью оператора плюс:

```
fruits = ["apples", "grapes", "oranges"]
veggies = ["corn", "kale", "mushrooms"]
grocery_list = fruits + veggies
grocery_list
```

```
['apples', 'grapes', 'oranges', 'corn', 'kale', 'mushrooms']
```

Учитывая список в качестве аргумента, `len()` возвращает длину списка или количество содержащихся в нем объектов:

```
numbers = [1, 2, 3, 200]
len(numbers)
```

```
4
```

Наиболее часто используемые методы:

`.append()` принимает объект в качестве аргумента и добавляет его в конец базового списка:

```
fruits = ["apples", "grapes", "oranges"]
fruits.append("blueberries")
fruits
```

```
['apples', 'grapes', 'oranges', 'blueberries']
```

`.sort()` сортирует базовый список на месте:

```
fruits.sort()
fruits
```

```
['apples', 'blueberries', 'grapes', 'oranges']
```

`.pop()` принимает в качестве аргумента целочисленный индекс, затем удаляет и возвращает элемент по этому индексу в базовом списке:

```
numbers_list = [1, 2, 3, 200]
numbers_list.pop(2)
```

```
3
```

```
numbers_list
```

```
[1, 2, 200]
```

> **Warning**
>
> Списки - это довольно распространенные и универсальные структуры данных в Python. Они настолько популярны, что разработчики иногда склонны злоупотреблять ими, что может сделать код неэффективным.

Операции над кортежами

Как и в случае со списками, вы также можете выполнять индексацию и нарезку кортежей:

```
employee = ("Jane", "Doe", 31, "Software Developer")
employee[0]
```

```
'Jane'
```

```
employee[1:3]
```

```
('Doe', 31)
```

Поскольку кортежи представляют собой последовательности, вы можете использовать индексы для извлечения определенных элементов в кортежах. Обратите внимание, что вы также можете получить фрагменты из кортежа с помощью операции нарезки.

Вы также можете добавить два кортежа с помощью оператора конкатенации:

```
first_tuple = (1, 2)
second_tuple = (3, 4)
first_tuple + second_tuple
```

```
(1, 2, 3, 4)
```

Операция конкатенации с двумя кортежами создает новый кортеж, содержащий все элементы в двух входных кортежах.

Как и в случае со списками и строками, вы можете использовать некоторые встроенные функции для управления кортежами. Например, `len()` возвращает длину кортежа или количество содержащихся в нем элементов:

```
numbers = (1, 2, 3)
len(numbers)
```

```
3
```

С кортежем в качестве аргумента `list()` возвращает список со всеми элементами входного кортежа:

```
numbers = (1, 2, 3)
list(numbers)
```

```
[1, 2, 3]
```

Поскольку кортежи являются неизменяемыми последовательностями, многие методы, доступные для списков, не работают с кортежами. Однако кортежи имеют два встроенных метода:

- `.count()`
- `.index()`

`.count()` принимает объект в качестве аргумента и возвращает количество раз, когда элемент появляется в базовом кортеже. Если объекта нет в кортеже, то `.count()` возвращает 0:

```
letter = ("a", "b", "b", "c", "a")
letter.count("a")
```

```
2
```

```
letter.count("c")
```

```
1
```

```
letter.count("d")
```

```
0
```

`.index()` принимает объект в качестве аргумента и возвращает индекс первого экземпляра этого объекта в имеющемся кортеже. Если объекта нет в кортеже, то `.index()` вызывает `ValueError`:

```
letter.index("a")
```

```
0
```

```
letter.index("c")
```

```
3
```

```
letter.index("d")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-93-5e53efde6b5c> in <module>
----> 1 letter.index("d")

ValueError: tuple.index(x): x not in tuple
```

Кортежи - довольно полезные структуры данных. Они эффективно используют память, неизменяемы и обладают большим потенциалом для управления данными, которые не должны изменяться пользователем. Их также можно использовать в качестве ключей словаря, о чем вы узнаете в следующем разделе.

Операции над словарями

Вы можете получить значение, связанное с ключом словаря, используя следующий синтаксис:

```
person1 = {"name": "John Doe", "age": 25, "job": "Python Developer"}
person1["name"]
```

```
'John Doe'
```

```
person1["age"]
```

```
25
```

Это очень похоже на операцию индексирования, но на этот раз вы используете ключ вместо индекса.

Вы также можете получить ключи, значения и пары ключ-значение в словаре с помощью `.keys()`, `.values()` и `.items()` соответственно:

```
# Получить все ключи
person1.keys()
```

```
dict_keys(['name', 'age', 'job'])
```

```
# Получить все значения
person1.values()
```

```
dict_values(['John Doe', 25, 'Python Developer'])
```

```
# Получить все пары ключ-значение
person1.items()
```

```
dict_items([('name', 'John Doe'), ('age', 25), ('job', 'Python Developer')])
```

Эти три метода являются фундаментальными инструментами, когда дело доходит до управления словарями в Python, особенно когда вы выполняете итерацию по словарю.

Операции над множествами

Вы можете использовать некоторые встроенные функции с множествами, как вы это делали с другими встроенными структурами данных. Например, если вы передадите множество в качестве аргумента функции `len(), вы получите количество элементов в множестве:

```
employees1 = {"John", "Jane", "Linda"}
len(employees1)
```

```
3
```

Вы также можете использовать операторы для управления множествами в Python. В этом случае большинство операторов представляют собой типичные операции над множеством, такие как объединение (|), пересечение (&), разность (-) и так далее:

```
primes = {2, 3, 5, 7}
evens = {2, 4, 6, 8}
```

```
# Объединение
primes | evens
```

```
{2, 3, 4, 5, 6, 7, 8}
```

```
# Пересечение
primes & evens
```

```
{2}
```

```
# Разница
primes - evens
```

```
{3, 5, 7}
```

Множества предоставляют набор методов, в том числе методы, которые выполняют операции с множествами, подобные тем, что в приведенном выше примере. Они также предоставляют методы для изменения или обновления базового множества.

`.add()` берет объект и добавляет его в множество:

```
primes = {2, 3, 5, 7}
primes.add(11)
primes
```

```
{2, 3, 5, 7, 11}
```

`.remove()` берет объект и удаляет его из множества:

```
primes = {2, 3, 5, 7, 11}
primes.remove(11)
primes
```

```
{2, 3, 5, 7}
```

Control Flow

Условные выражения

Иногда вам нужно запустить (или не запускать) данный блок кода в зависимости от того, выполняются ли определенные условия. В этом случае условные выражения - ваш союзник. Эти операторы управляют выполнением группы операторов на основе значения истинности выражения. Вы можете создать условный оператор в Python с ключевым словом `if` и следующим общим синтаксисом:

```
if expr0:
    # Run if expr0 is true
    # Your code goes here...
elif expr1:
    # Run if expr1 is true
    # Your code goes here...
elif expr2:
    # Run if expr2 is true
    # Your code goes here...
...
else:
    # Run if all expressions are false
    # Your code goes here...

# Next statement
```

Оператор `if` запускает только один блок кода. Другими словами, если `expr0` истинно, будет выполняться только связанный с ним блок кода. После этого выполнение переходит к оператору непосредственно под оператором `if`.

Первое предложение `elif` оценивает `expr1` только в том случае, если `expr0` ложно. Если `expr0` - `False`, а `expr1` - `True`, то будет выполняться только блок кода, связанный с `expr1`, и так далее. Предложение `else` является необязательным и будет выполняться только в том случае, если все ранее оцененные условия ложны. У вас может быть столько предложений `elif`, сколько вам нужно, включая ни одного, но у вас может быть только одно предложение `else`.

Вот несколько примеров того, как это работает:

```
age = 21
if age >= 18:
    print("Вы совершеннолетний")
```

```
Вы совершеннолетний
```

```
age = 16
if age >= 18:
    print("Вы совершеннолетний")
else:
    print("Вы НЕсовершеннолетний")
```

```
Вы НЕсовершеннолетний
```

```
age = 18
if age > 18:
    print("Вы старше 18 лет")
elif age == 18:
    print("Вам ровно 18 лет")
```

```
Вам ровно 18 лет
```

В первом примере возраст равен `21`, поэтому условие истинно, и Python выводит на экран «Вы совершеннолетний». Во втором примере выражение `age >= 18` оценивается как `False`, поэтому Python запускает блок кода предложения `else` и печатает на экране «Вы НЕсовершеннолетний».

В последнем примере первое выражение `age > 18` ложно, поэтому выполнение переходит к предложению `elif`. Условие в этом предложении истинно, поэтому Python запускает связанный блок кода и печатает «Вам ровно 18 лет».

Тернарный оператор

Синтаксис тернарного оператора:

```
<expr1> if <conditional_expr> else <expr2>
```

Он отличается от оператора `if`, потому что это не структура управления, которая направляет поток выполнения программы. Он действует больше как оператор, определяющий выражение. В приведенном выше примере сначала вычисляется `<conditional_expr>`. Если это правда, выражение оценивается как `<expr1>`. Если это ложь, выражение оценивается как `<expr2>`.

Обратите внимание на неочевидный порядок: сначала вычисляется среднее выражение, и на основе этого результата возвращается одно из выражений на концах. Вот несколько примеров, которые, надеюсь, помогут прояснить ситуацию:

```
raining = False
print("Let's go to the", 'beach' if not raining else 'library')
```

```
Let's go to the beach
```

```
raining = True
print("Let's go to the", 'beach' if not raining else 'library')
```

```
Let's go to the library
```

```
age = 12
'minor' if age < 21 else 'adult'
```

```
'minor'
```

```
'yes' if ('qux' in ['foo', 'bar', 'baz']) else 'no'
```

```
'no'
```

> **Note**
>
> Условное выражение Python похоже на `<conditional_expr> ? <expr1> : синтаксис <expr2>`, используемый многими другими языками, включая C, Perl и Java.
>
> Вы можете видеть в PEP 308, что `<conditional_expr> ? <expr1> : синтаксис <expr2>` рассматривался для Python, но в конечном итоге был отклонен в пользу синтаксиса, показанного выше.

Условные выражения также используют вычисление по сокращённой схеме. Части условного выражения не вычисляются, если в этом нет необходимости.

В выражении `<expr1> if <conditional_expr> else <expr2>`:

- Если `<conditional_expr>` истинно, возвращается `<expr1>`, а `<expr2>` не вычисляется.
- Если `<conditional_expr>` ложно, возвращается `<expr2>`, а `<expr1>` не вычисляется.

Вы можете проверить это, используя выражения, которые могут вызвать ошибку:

```
'foo' if True else 1/0
```

```
'foo'
```

```
1/0 if False else 'bar'
```

```
'bar'
```

В обоих случаях `1/0` не вычисляется, поэтому исключение не возникает.

Циклы

Если вам нужно повторить фрагмент кода несколько раз, чтобы получить окончательный результат, вам может потребоваться цикл. Циклы - это распространенный способ многократного повторения и выполнения определенных действий на каждой итерации. Python предоставляет два типа циклов:

- циклы `for` для определенной итерации или выполнения заданного числа повторений
- циклы `while` для неопределенной итерации или повторение до тех пор, пока не будет выполнено заданное условие

Цикл for

```
for loop_var in iterable:
    # Этот блок кода повторяется до тех пор, пока iterable не будет закончен
    # Делаем что-нибудь с loop_var...
    if break_condition:
        break  # Выйти из цикла
    if continue_condition:
        continue  # Продолжить цикл без выполнения оставшегося кода
    # Оставшийся код...

# Следующее выражение
```

Этот тип цикла выполняет столько итераций, сколько элементов в `iterable`. Обычно вы используете каждую итерацию для выполнения операции над значением `loop_var`. Операторы `break` и `continue` являются необязательными.

```
for i in (1, 2, 3, 4, 5):
    print(i)
else:
    print("The loop wasn't interrupted")
```

```
1
2
3
4
5
The loop wasn't interrupted
```

Когда цикл обрабатывает последнее число в кортеже, поток выполнения переходит в предложение `else` и печатает *«The loop wasn't interrupted»*. Это потому, что ваш цикл не был прерван оператором `break`. Вы обычно используете предложение `else` в циклах, в блоке кода которых есть оператор `break`. В противном случае в этом нет необходимости.

break

Если в цикле встречается `break_condition`, то оператор `break` прерывает выполнение цикла и переходит к следующему оператору ниже цикла, не используя остальные элементы в итерации:

```
number = 3
for i in (1, 2, 3, 4, 5):
    if i == number:
        print("Number found:", i)
        break
else:
    print("Number not found")
```

```
Number found: 3
```

Когда `i == 3`, цикл печатает *Number found: 3*, а затем выполняет `break`. Это прерывает цикл, и выполнение переходит к строке ниже цикла без выполнения предложения else. Если вы установите `number` равным `6` или любым другим числом, не входящим в набор чисел, то цикл не попадет в оператор `break` и напечатает *Number not found*.

continue

Если цикл достигает `continue_condition`, то оператор `continue` возобновляет цикл без выполнения кода в блоке цикла:

```
for i in (1, 2, 3, 4, 5):
    if i == 3:
        continue
    print(i)
```

```
1
2
4
5
```

На этот раз инструкция `continue` перезапускает цикл, когда `i == 3`. Вот почему вы не видите цифру `3` в выводе.

Оба оператора, `break` и `continue`, должны быть заключены в условное выражение. В противном случае цикл всегда прерывается, когда он выполняет `break`, и продолжается, когда он выполняет `continue`.

Цикл while

Обычно цикл `while` используется, если заранее неизвестно, сколько итераций вам нужно для выполнения операции. Вот почему этот цикл используется для выполнения неопределенных итераций.

```
while expression:
    # Этот блок кода повторяется до тех пор, пока expression истинно
    # Делаем что-нибудь...
    if break_condition:
        break  # Выйти из цикла
    if continue_condition:
        continue  # Продолжить цикл без выполнения оставшегося кода
    # Оставшийся код...

# Следующее выражение
```

Этот цикл работает аналогично циклу `for`, но он будет повторяться до тех пор, пока выражение не станет ложным. Распространенная проблема с этим типом цикла возникает, когда вы предоставляете выражение, которое никогда не оценивается как `False`. В этом случае цикл будет повторяться бесконечно.

Вот пример того, как работает цикл `while`:

```
count = 1
while count < 5:
    print(count)
    count = count + 1
else:
    print("The loop wasn't interrupted")
```

```
1
2
3
4
The loop wasn't interrupted
```

Предложение `else` является необязательным, и его обычно используют с оператором `break` в блоке кода цикла. Здесь `break` и `continue` работают так же, как в цикле `for`.

Бывают ситуации, когда нужен бесконечный цикл. Например, приложения с графическим интерфейсом пользователя работают в бесконечном цикле, который управляет пользовательскими событиями. Этому циклу требуется оператор `break`, чтобы завершить цикл, когда, например, пользователь выходит из приложения. В противном случае приложение продолжало бы работать вечно.

Обработка ошибок

Исключения и синтаксические ошибки

Программа Python завершается, как только обнаруживает ошибку. В Python ошибка может быть **синтаксической ошибкой** или **исключением**.

Синтаксические ошибки возникают, когда парсер обнаруживает неверный оператор. Обратите внимание на следующий пример:

```
print( 0 / 0 ))
```

```
  File "<ipython-input-14-c3931f671051>", line 1
    print( 0 / 0 ))
                   ^
SyntaxError: unmatched ')'
```

Стрелка указывает, где парсер обнаружил синтаксическую ошибку. В этом примере лишняя скобка. Удалите ее и снова запустите свой код:

```
print( 0 / 0 )
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-21-f5008b880c58> in <module>
----> 1 print( 0 / 0 )

ZeroDivisionError: division by zero
```

На этот раз вы столкнулись с ошибкой исключения. Этот тип ошибки возникает всякий раз, когда синтаксически правильный код Python приводит к ошибке. В последней строке сообщения указано, с каким типом ошибки исключения вы столкнулись.

Вместо того, чтобы показывать ошибку исключения сообщения, Python детализирует, какой тип ошибки исключения был обнаружен. В данном случае это была ошибка `ZeroDivisionError`. В Python существуют различные встроенные исключения, а также возможность создавать самоопределяемые исключения.

Вызов исключения

Мы можем использовать вызов исключения, чтобы сгенерировать исключение при каком-то условии.

```
x = 10
if x > 5:
    raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-23-3dcc1a597936> in <module>
      1 x = 10
      2 if x > 5:
----> 3     raise Exception('x should not exceed 5. The value of x was:
{}'.format(x))

Exception: x should not exceed 5. The value of x was: 10
```

Программа останавливается и отображает наше исключение на экране, предлагая подсказки о том, что пошло не так.

Обработка исключений

try/except

Блок `try` и `except` в Python используется для перехвата и обработки исключений. Python выполняет код, следующий за оператором `try`, как «нормальную» часть программы. Код, следующий за оператором `except`, является ответом программы на любые исключения в предыдущем блоке `try`.

Как вы видели ранее, когда синтаксически правильный код сталкивается с ошибкой, Python выдает ошибку исключения. Эта ошибка исключения приведет к сбою программы, если она не будет обработана. Предложение `except` определяет, как ваша программа реагирует на исключения.

```
import sys
assert ('linux' in sys.platform), "Function can only run on Linux systems."
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-27-98b58e557371> in <module>
      1 import sys
----> 2 assert ('linux' in sys.platform), "Function can only run on Linux systems."

AssertionError: Function can only run on Linux systems.
```

`'linux' in sys.platform` вернет True только в системе `Linux`. Если вы попытаетесь выполнить код в операционной системе, отличной от `Linux`, произойдет исключение `AssertionError`.

```
try:
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')
except AssertionError as error:
    print(error)
    print('Linux function was not executed')
```

В коде выше мы обработали исключение. Если бы вы запустили этот код на машине с `Windows`, вы бы получили следующий результат:

```
Function can only run on Linux systems.
Linux function was not executed
```

> **Warning**
>
> Перехват исключений скрывает все ошибки... даже совершенно неожиданные. Вот почему вам следует избегать голых исключений в ваших программах на Python. Вместо этого вам нужно обращаться к конкретным классам исключений, которые вы хотите перехватывать и обрабатывать.

- Предложение `try` выполняется до момента, когда встречается первое исключение.
- Внутри предложения `except` или обработчика исключения вы определяете, как программа реагирует на исключение.
- Вы можете предвидеть множественные исключения и различать, как программа должна на них реагировать.
- Избегайте использования заглушек в блоке `except`.

else

В Python с помощью оператора `else` вы можете указать программе выполнить определенный блок кода только при отсутствии исключений.

Взгляните на следующий пример:

```
try:
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')
except AssertionError as error:
    print(error)
else:
    print('Executing the else clause.')
```

```
Function can only run on Linux systems.
```

Если бы вы запустили этот код в системе `Linux`, результат был бы следующим:

```
Doing something.
Executing the else clause.
```

Поскольку программа не столкнулась с какими-либо исключениями, было выполнено предложение `else`. Вы также можете попробовать запустить код внутри предложения `else` и отловить там возможные исключения:

```
try:
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
```

```
Function can only run on Linux systems.
```

Если бы вы выполнили этот код на машине `Linux`, вы бы получили следующий результат:

```
Doing something.
[Errno 2] No such file or directory: 'file.log'
```

Из выходных данных вы можете видеть, что первый блок `try` был запущен. Поскольку никаких исключений не обнаружено, была сделана попытка открыть `file.log`. Этот файл не существует, и вместо его открытия вы поймали исключение `FileNotFoundError`.

finally
Представьте, что вам всегда бы приходилось выполнять какое-то действие для очистки после выполнения вашего кода. Python позволяет сделать это с помощью предложения `finally`.

Взгляните на следующий пример:

```
try:
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('Cleaning up, irrespective of any exceptions.')
```

```
Function can only run on Linux systems.
Cleaning up, irrespective of any exceptions.
```

В коде выше всё, что указано в предложении finally, будет выполнено. Не имеет значения, встретите ли вы исключение где-нибудь в блоках `try` или `else`.

Выполнение этого кода на компьютере с `Windows` приведет к следующему:

```
Function can only run on Linux systems.
Cleaning up, irrespective of any exceptions.
```

Functions & Modules

Повторное использование кода - очень важная часть программирования на любом языке. Увеличение размера кода затрудняет поддержку.

> **Interesting info**
>
> Чтобы большой программный проект был успешным, важно придерживаться принципа «**Don't Repeat Yourself**» (Не Повторяйся) или «**DRY**».

Мы уже рассмотрели один способ избавиться от повторения кода: с помощью циклов. В этом модуле мы рассмотрим еще два: функции и модули.

Функции

Любой оператор, состоящий из слова, за которым следует информация в круглых скобках, является вызовом функции.

Вот несколько примеров, которые вы уже видели:

```
print('Hello World!')
len([1, 2, 3, 4])
str(12)
```

Слова перед круглыми скобками - это имена функций, а значения, разделенные запятыми внутри круглых скобок, - аргументы функции.

Помимо использования предопределенных функций, вы можете создавать свои собственные функции с помощью оператора `def`.

```
def my_func():
    print('spam')
    print('spam')
    print('spam')

my_func()
```

```
spam
spam
spam
```

Она не принимает аргументов и трижды печатает «spam». Она определяется, а затем вызывается. Операторы в функции выполняются только при ее вызове. Блок кода внутри каждой функции начинается с двоеточия (:) и имеет отступ.

Вы должны определить функции до их вызова, точно так же, как вы должны назначать переменные перед их использованием:

```
hello()

def hello():
    print('hello')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-84ea1bbbea4b> in <module>
----> 1 hello()
      2
      3 def hello():
      4     print('hello')

NameError: name 'hello' is not defined
```

Аргументы

Все определения функций, которые мы рассмотрели до сих пор, были функциями с нулевыми аргументами, которые вызываются с пустыми круглыми скобками. Однако большинство функций принимают аргументы.

В приведенном ниже примере определяется функция, которая принимает один аргумент:

```
def print_with_exclamation(word):
    print(word + '!')

print_with_exclamation('python')
```

```
python!
```

Вы также можете определять функции с более чем одним аргументом; разделите их запятыми:

```
def print_sum_twice(x, y):
    print(x + y)
    print(x + y)

print_sum_twice(5, 8)
```

```
13
13
```

Аргументы функции могут использоваться как переменные внутри определения функции. Однако на них нельзя ссылаться вне определения функции. Это также относится к другим переменным, созданным внутри функции.

```
def func(variable):
    variable += 1
    print(variable)

func(7)
print(variable)
```

```
8
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-8f2a1a4178d6> in <module>
      4
      5 func(7)
----> 6 print(variable)

NameError: name 'variable' is not defined
```

Технически параметры - это переменные в определении функции, а аргументы - это значения, помещаемые в параметры при вызове функций.

Return

Некоторые функции, такие как `int` или `str`, возвращают значение, которое можно использовать позже. Чтобы сделать это для ваших определенных функций, вы можете использовать оператор `return`.

```
def max(x, y):
    if x >= y:
        return x
    else:
        return y

print(max(4, 7))
z = max(8, 5)
print(z)
```

```
7
8
```

Оператор `return` не может использоваться вне определения функции.

Как только вы возвращаете значение из функции, она немедленно перестает выполняться. Никакого кода после оператора возврата никогда не произойдет.

```
def add_numbers(x, y):
    total = x + y
    return total
    print("This won't be printed")

print(add_numbers(4, 5))
```

```
9
```

Функции как объекты

Хотя они создаются не так, как обычные переменные, функции такие же, как и любые другие значения. Они могут быть назначены и переназначены переменным, а затем на них будут ссылаться эти имена.

```
def multiply(x, y):
    return x * y

a = 4
b = 7
operation = multiply
print(operation(a, b))
```

```
28
```

В приведенном выше примере функция умножения назначается переменной `operation`. Теперь `operation` также можно использовать для вызова функции.

Функции также могут использоваться в качестве аргументов других функций.

```
def add(x, y):
    return x + y

def do_twice(func, x , y):
    return func(func(x, y), func(x, y))

print(do_twice(add, 5, 10))
```

```
30
```

Как видите, функция `do_twice` принимает функцию в качестве аргумента и вызывает ее в своем теле.

Лямбды

При обычном создании функции (с использованием `def`) она автоматически присваивается переменной. Это отличается от создания других объектов, таких как строки и целые числа, которые могут быть созданы «на лету», без присвоения их переменной.

То же самое возможно и с функциями, при условии, что они созданы с использованием лямбда-синтаксиса. Созданные таким образом функции называются анонимными.

Этот подход чаще всего используется при передаче простой функции в качестве аргумента другой функции. Синтаксис показан в следующем примере и состоит из ключевого слова `lambda`, за которым следует список аргументов, двоеточие и выражение.

```
def my_func(f, arg):
    return f(arg)

my_func(lambda x: 2 * x * x, 5)
```

```
50
```

> **Interesting info**
>
> Лямбда-функции получили свое название от лямбда-исчисления, которое представляет собой модель вычислений, изобретенную Алонзо Черчем.

Лямбда-функции не так мощны, как именованные. Они могут делать только то, что требует одного выражения - обычно эквивалентно одной строке кода.

```
# named function
def polynomial(x):
    return x ** 2 + 5 * x + 4
print(polynomial(-4))

# lambda
print((lambda x:  x ** 2 + 5 * x + 4)(-4))
```

```
0
0
```

В приведенном выше коде мы на лету создали анонимную функцию и вызвали ее с аргументом.

Лямбда-функции можно назначать переменным и использовать как обычные функции.

```
double = lambda x: x * 2
print(double(7))
```

```
14
```

Однако для этого редко есть веская причина - обычно лучше определить функцию с помощью `def`.

map & filter

Встроенные функции `map` и `filter` - очень полезные функции высшего порядка, которые работают со списками (или подобными объектами, называемыми `iterables`).

`map` принимает функцию и итерацию в качестве аргументов и возвращает новую итерацию с функцией, примененной к каждому аргументу.

```
def add_five(x):
    return x + 5

nums = [11, 22, 33, 44, 55]
result = list(map(add_five, nums))
print(result)
```

```
[16, 27, 38, 49, 60]
```

Мы можем легче достичь того же результата, используя лямбды.

```
nums = [11, 22, 33, 44, 55]
result = list(map(lambda x: x + 5, nums))
print(result)
```

```
[16, 27, 38, 49, 60]
```

Чтобы преобразовать результат в список, мы явно использовали `list`.

`filter` фильтрует итерацию, удаляя элементы, не соответствующие предикату (функция, возвращающая логическое значение).

```
nums = [11, 22, 33, 44, 55]
result = list(filter(lambda x: x % 2 == 0, nums))
print(result)
```

```
[22, 44]
```

Как и `map`, результат должен быть явно преобразован в список, если вы хотите его распечатать.

Генераторы

Генераторы - это тип итераторов, таких как списки или кортежи. В отличие от списков, они не позволяют индексацию с произвольными индексами, но их все же можно перебирать с помощью циклов `for`.

Их можно создать с помощью функций и оператора `yield`.

```
def countdown():
    i = 5
    while i > 0:
        yield i
        i -= 1
for i in countdown():
    print(i)
```

```
5
4
3
2
1
```

Оператор `yield` используется для определения генератора, заменяя возврат функции для предоставления результата без разрушения локальных переменных.

Из-за того, что они выдают по одному элементу за раз, генераторы не имеют ограничений памяти списков. На самом деле они могут быть **бесконечными**!

```
def infinite_sevens():
    while True:
        yield 7

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

Короче говоря, генераторы позволяют объявлять функцию, которая ведет себя как итератор, то есть ее можно использовать в цикле `for`.

Конечные генераторы можно преобразовать в списки, передав их в качестве аргумента функции `list`.

```
def numbers(x):
    for i in range(x):
        if i % 2 == 0:
            yield i
print(list(numbers(11)))
```

```
[0, 2, 4, 6, 8, 10]
```

Использование генераторов приводит к повышению производительности, что является результатом ленивой (по запросу) генерации значений, что приводит к снижению использования памяти. Кроме того, нам не нужно ждать, пока все элементы будут сгенерированы, прежде чем мы начнем их использовать.

Декораторы

Декораторы позволяют изменять функции с помощью других функций. Это идеально, когда вам нужно расширить функциональность функций, которые вы не хотите изменять.

```
def decor(func):
    def wrap():
        print('============')
        func()
        print('============')
    return wrap

def print_text():
    print('Hello World!')

decorated = decor(print_text)
decorated()
```

```
============
Hello World!
============
```

Мы определили функцию с именем `decor`, которая имеет единственный параметр `func`. Внутри `decor` мы определили вложенную функцию с именем `wrap`. Функция `wrap` напечатает строку, затем вызовет функцию `func()` и напечатает другую строку. Функция `decor` возвращает в качестве результата функцию `wrap`.

Можно сказать, что переменная `decorated` - декорированная версия `print_text` - это `print_text` плюс еще что-то. Фактически, если бы мы написали полезный декоратор, мы могли бы полностью заменить `print_text` декорированной версией, чтобы мы всегда получали нашу версию `print_text` «плюс что-то».

Это делается путем переназначения переменной, содержащей нашу функцию:

```
print_text = decor(print_text)
print_text()
```

```
============
Hello World!
============
```

Теперь `print_text` соответствует нашей декорированной версии.

В нашем предыдущем примере мы декорировали нашу функцию, заменив переменную, содержащую функцию, на упакованную версию.

Этот шаблон можно использовать в любое время, чтобы обернуть любую функцию. Python обеспечивает поддержку обертывания функции в декораторе, предварительно добавляя определение функции с именем декоратора и символом `@`.

Если мы определяем функцию, мы можем «декорировать» ее символом `@`, например:

```
def decor(func):
    def wrap():
        print('============')
        func()
        print('============')
    return wrap

@decor
def print_text():
    print('Hello World!')

print_text()
```

```
============
Hello World!
============
```

Одна функция может иметь несколько декораторов.

Модули

Модули - это фрагменты кода, написанные другими людьми для выполнения общих задач, таких как генерация случайных чисел, выполнение математических операций и т. д.

Основной способ использования модуля - это добавить `import module_name` вверху вашего кода, а затем использовать `module_name.var` для доступа к функциям и значениям с именем `var` в модуле. Например, в следующем примере модуль `random` используется для генерации случайных чисел:

```
import random

for i in range(5):
    value = random.randint(1, 6)
    print(value)
```

```
2
2
4
5
5
```

Код использует функцию `randint`, определенную в модуле `random`, для печати 5 случайных чисел в диапазоне от 1 до 6.

Есть еще один вид импорта, который можно использовать, если вам нужны только определенные функции из модуля.
Они принимают форму `from module_name import var`, а затем var можно использовать, как если бы он был определен в вашем коде обычным образом.
Например, чтобы импортировать только константу `pi` из модуля `math`:

```
from math import pi

print(pi)
```

```
3.141592653589793
```

Используйте список, разделенный запятыми, для импорта нескольких объектов:

```
from math import pi, sqrt
```

`*` импортирует все объекты из модуля. Например: `from math import *`
Обычно это не рекомендуется, поскольку при этом путают переменные в вашем коде с переменными во внешнем модуле.

Попытка импортировать недоступный модуль вызывает ошибку `ImportError`.

```
import some_module
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-42-beceebbbf090> in <module>
----> 1 import some_module

ModuleNotFoundError: No module named 'some_module'
```

Вы можете импортировать модуль или объект под другим именем, используя ключевое слово `as`. Это в основном используется, когда модуль или объект имеет длинное или непонятное имя.

```
from math import sqrt as square_root

print(square_root(100))
```

```
10.0
```

Стандартная библиотека и pip

В Python есть три основных типа модулей: те, которые вы пишете сами, те, которые вы устанавливаете из внешних источников, и те, которые предустановлены вместе с Python.
Последний тип называется стандартной библиотекой и содержит множество полезных модулей. Некоторые из полезных модулей стандартной библиотеки включают `string`, `re`, `datetime`, `math`, `random`, `os`, `multiprocessing`, `subprocess`, `socket`, `email`, `json`, `doctest`, `unittest`, `pdb`, `argparse` и `sys`.

Задачи, которые может выполнять стандартная библиотека, включают синтаксический анализ строк, сериализацию данных, тестирование, отладку и управление датами, сообщениями электронной почты, аргументами командной строки и многое другое!

Обширная стандартная библиотека Python - одна из его основных сильных сторон как языка.

Некоторые модули в стандартной библиотеке написаны на Python, а некоторые - на C.
Большинство из них доступны на всех платформах, но некоторые относятся к Windows или Unix.

Многие сторонние модули Python хранятся в индексе пакетов Python (**PyPI**).
Лучший способ установить их - использовать программу `pip`. Он устанавливается по умолчанию в современных дистрибутивах Python. Если у вас его нет, его легко установить онлайн. После того, как он у вас есть, установить библиотеки из PyPI очень просто. Найдите имя библиотеки, которую вы хотите установить, перейдите в командную строку и введите `pip install <library_name>`. Как только вы это сделаете, импортируйте библиотеку и используйте ее в своем коде.

Важно вводить команды `pip` в командной строке, а не в интерпретаторе Python.

Object-oriented programming (OOP)

Классы

Ранее мы рассмотрели две парадигмы программирования - **императивную** (с использованием операторов, циклов и функций в качестве подпрограмм) и **функциональную** (с использованием чистых функций, функций высшего порядка).

Еще одна очень популярная парадигма - **объектно-ориентированное программирование** (ООП).
Объекты создаются с использованием **классов**, которые на самом деле являются фокусом ООП. Класс описывает, каким будет объект, но отделен от самого объекта. Другими словами, класс можно описать как план, описание или определение объекта.
Вы можете использовать один и тот же класс в качестве чертежа для создания нескольких различных объектов.

Классы создаются с использованием ключевого слова **class** и блока с отступом, который содержит **методы класса** (которые являются функциями). Ниже приведен пример простого класса и его объектов.

```
class Cat:
    def __init__(self, color, legs):
        self.color = color
        self.legs = legs

felix = Cat("ginger", 4)
rover = Cat("dog-colored", 4)
stumpy = Cat("brown", 3)
```

Этот код определяет класс с именем `Cat`, который имеет два атрибута: `color` и `legs`.
Затем класс используется для создания 3-х отдельных объектов этого класса.

`__init__`

Метод `__init__` - самый важный метод в классе.
Он вызывается при создании экземпляра (объекта) класса с использованием имени класса в качестве функции.

Все методы должны иметь в качестве первого параметра `self`. Хотя он явно не передается, Python добавляет аргумент `self` в список за вас; вам не нужно включать его при вызове методов. В определении метода `self` относится к экземпляру, вызывающему метод.

Экземпляры класса имеют **атрибуты**, которые представляют собой связанные с ними фрагменты данных.
В этом примере экземпляры `Cat` имеют атрибуты `color` и `legs`. К ним можно получить доступ, поставив точку и имя атрибута после экземпляра.
Таким образом, в методе `__init__` `self.attribute` может использоваться для установки начального значения атрибутов экземпляра.

```
class Cat:
    def __init__(self, color, legs):
        self.color = color
        self.legs = legs

felix = Cat("ginger", 4)
print(felix.color)
```

```
ginger
```

В приведенном выше примере метод `__init__` принимает два аргумента и назначает их атрибутам объекта. Метод `__init__` называется **конструктором класса**.

Методы

Классы могут иметь другие **методы**, определенные для добавления к ним функциональности.
Помните, что все методы должны иметь в качестве первого параметра `self`.
Доступ к этим методам осуществляется с использованием того же синтаксиса точек, что и для атрибутов.

```
class Dog:
    def __init__(self, color, name):
        self.color = color
        self.name = name

    def bark(self):
        print('Woof!')

fido = Dog('Fido', 'brown')
print(fido.name)
fido.bark()
```

```
brown
Woof!
```

Классы также могут иметь **атрибуты класса**, созданные путем присвоения переменных в теле класса. К ним можно получить доступ либо из экземпляров класса, либо из самого класса.

```
class Dog:
    legs = 4
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def bark(self):
        print('Woof!')

fido = Dog('Fido', 'brown')
print(fido.legs)
print(Dog.legs)
```

```
4
4
```

Атрибуты класса являются общими для всех экземпляров класса.

Попытка получить доступ к атрибуту экземпляра, который не определен, вызывает ошибку `AttributeError`. Это также происходит, когда вы вызываете неопределенный метод.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

rect = Rectangle(7, 8)
print(rect.color)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-5-f94adae8ff8b> in <module>
      5
      6 rect = Rectangle(7, 8)
----> 7 print(rect.color)

AttributeError: 'Rectangle' object has no attribute 'color'
```

Наследование

Наследование дает возможность разделять функциональные возможности между классами.
Представьте себе несколько классов: `Cat`, `Dog`, `Rabbit` и так далее. Хотя они могут отличаться в некоторых отношениях (только `Dog` может иметь метод `bark`), они, вероятно, будут похожи в других (все имеют атрибуты `color` и `name`).
Это сходство может быть выражено, если все они унаследованы от **суперкласса** `Animal`, который содержит общие функции.
Чтобы унаследовать класс от другого класса, поместите имя суперкласса в круглые скобки после имени класса.

```
class Animal:
    def __init__(self, name, color):
        self.name = name
        self.color = color

class Cat(Animal):
    def purr(self):
        print('Purr...')

class Dog(Animal):
    def bark(self):
        print('Woof!')

fido = Dog('Fido', 'brown')
print(fido.color)
fido.bark()
```

```
brown
Woof!
```

Класс, который наследуется от другого класса, называется **подклассом**. Класс, от которого другой класс унаследован, называется **суперклассом**. Если класс наследуется от другого с такими же атрибутами или методами, он переопределяет их.

```
class Wolf:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def bark(self):
        print('Grr...')

class Dog(Wolf):
    def bark(self):
        print('Woof!')

husky = Dog('Max', 'grey')
husky.bark()
```

```
Woof!
```

В приведенном выше примере `Wolf` - это суперкласс, `Dog` - подкласс.

Наследование также может быть косвенным. Один класс может наследоваться от другого, и этот класс может наследоваться от третьего класса.

```
class A:
    def method(self):
        print('A method')

class B(A):
    def another_method(self):
        print('B method')

class C(B):
    def third_method(self):
        print('C method')

c = C()
c.method()
c.another_method()
c.third_method()
```

```
A method
B method
C method
```

Однако циклическое наследование невозможно.

Функция `super` - это полезная функция, связанная с наследованием, которая обращается к родительскому классу. Его можно использовать для поиска метода с определенным именем в суперклассе объекта.

```
class A:
    def spam(self):
        print(1)

class B(A):
    def spam(self):
        print(2)
        super().spam()

B().spam()
```

```
2
1
```

`super().spam()` вызывает метод `spam` суперкласса.

Магические методы (dunders) и перегрузка операторов

Магические методы - это специальные методы, у которых в начале и в конце названия есть двойное подчеркивание. Они также известны как **dunders**. Пока что мы столкнулись только с `__init__`, но есть и другие. Они используются для создания функциональности, которую нельзя представить как обычный метод.

Одним из распространенных способов их использования является **перегрузка оператора**. Это означает определение операторов для пользовательских классов, которые позволяют использовать с ними такие операторы, как `+` и `*`.

Примером магического метода является `__add__` для `+`.

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

first = Vector2D(5, 7)
second = Vector2D(3, 9)

result = first + second
print(result.x)
print(result.y)
```

```
8
16
```

Метод `__add__` позволяет определять настраиваемое поведение для оператора `+` в нашем классе.
Как видите, он добавляет соответствующие атрибуты объектов и возвращает новый объект, содержащий результат.
Как только он определен, мы можем добавить два объекта класса вместе.

Дополнительные магические методы для общих операторов:

`__sub__` для -

`__mul__` для *

`__truediv__` для /

`__floordiv__` для //

`__mod__` для %

`__pow__` для **

`__and__` для &

`__xor__` для ^

`__or__` для |

Выражение `x + y` переводится в `x.__add__(y)`.
Однако, если `x` не реализовал `__add__`, а `x` и `y` имеют разные типы, то вызывается `y.__radd__(x)`.

Для всех только что упомянутых магических методов существуют эквивалентные **r-методы**.

```
class SpecialString:
    def __init__(self, cont):
        self.cont = cont

    def __truediv__(self, other):
        line = '=' * len(other.cont)
        return '\n'.join([self.cont, line, other.cont])

spam = SpecialString('spam')
hello = SpecialString('Hello World!')
print(spam / hello)
```

```
spam
============
Hello World!
```

В приведенном выше примере мы определили **операцию деления** для нашего класса `SpecialString`.

Python также предоставляет магические методы для сравнения.

`__lt__` для <

`__le__` для <=

`__eq__` для ==

`__ne__` для !=

`__gt__` для >

`__ge__` для> =

Если `__ne__` не реализован, он возвращает противоположность `__eq__`. Других отношений между другими операторами нет.

```
class SpecialString:
    def __init__(self, cont):
        self.cont = cont

    def __gt__(self, other):
        for index in range(len(other.cont) + 1):
            result = other.cont[:index] + '>' + self.cont
            result += '>' + other.cont[index:]
            print(result)

spam = SpecialString('spam')
eggs = SpecialString('eggs')
spam > eggs
```

```
>spam>eggs
e>spam>ggs
eg>spam>gs
egg>spam>s
eggs>spam>
```

Как видите, вы можете определить любое настраиваемое поведение для перегруженных операторов.

Есть несколько волшебных способов заставить классы действовать как контейнеры.

`__len__` для `len()`

`__getitem__` для индексации

`__setitem__` для присвоения индексированным значениям

`__delitem__` для удаления индексированных значений

`__iter__` для итерации по объектам (например, в циклах `for`)

`__contains__` для `in`

Есть много других магических методов, которые мы здесь не будем рассматривать, например `__call__` для вызова объектов как функций и `__int__`, `__str__` и т.п. для преобразования объектов во встроенные типы.

```
import random

class VagueList:
    def __init__(self, cont):
        self.cont = cont

    def __getitem__(self, index):
        return self.cont[index + random.randint(-1, 1)]

    def __len__(self):
        return random.randint(0, len(self.cont) * 2)

vague_list = VagueList(['A', 'B', 'C', 'D', 'E'])
print(len(vague_list))
print(len(vague_list))
print(vague_list[2])
print(vague_list[2])
```

```
10
1
C
B
```

Мы переопределили функцию `len()` для класса `VagueList`, чтобы она возвращала случайное число.
Функция индексации также возвращает случайный элемент в диапазоне из списка на основе выражения.

Сокрытие данных

Ключевой частью объектно-ориентированного программирования является **инкапсуляция**, которая включает в себя упаковку связанных переменных и функций в один простой в использовании объект - экземпляр класса. Связанная концепция - это **скрытие данных**, в котором говорится, что детали реализации класса должны быть скрыты, а для тех, кто хочет использовать этот класс, должен быть представлен чистый стандартный интерфейс. В других языках программирования это обычно делается с помощью частных методов и атрибутов, которые блокируют внешний доступ к определенным методам и атрибутам в классе.

Философия Python немного отличается. Часто говорят, что «мы все здесь взрослые», что означает, что вы не должны вводить произвольные ограничения на доступ к частям класса. Следовательно, нет способов сделать метод или атрибут строго закрытым.

Однако есть способы отговорить людей от доступа к частям класса, например, указав, что это деталь реализации и что их следует использовать на свой страх и риск.

Слабо закрытые методы и атрибуты имеют в начале **один знак подчеркивания**.
Это сигнализирует о том, что они являются частными и не должны использоваться внешним кодом. Однако это в основном лишь соглашение и не мешает внешнему коду получить к ним доступ.
Его единственный фактический эффект заключается в том, что **from module_name import *** не импортирует переменные, которые начинаются с одного символа подчеркивания.

```
class Queue:
    def __init__(self, contents):
        self._hidden_list = contents

    def push(self, value):
        self._hidden_list.insert(0, value)

    def pop(self):
        return self._hidden_list.pop(-1)

    def __repr__(self):
        return f'Queue({self._hidden_list})'

queue = Queue([1, 2, 3])
print(queue)
queue.push(0)
print(queue)
queue.pop()
print(queue)
print(queue._hidden_list)
```

```
Queue([1, 2, 3])
Queue([0, 1, 2, 3])
Queue([0, 1, 2])
[0, 1, 2]
```

В приведенном выше коде атрибут `_hiddenlist` помечен как частный, но к нему по-прежнему можно получить доступ во внешнем коде. Магический метод `__repr__` используется для строкового представления экземпляра.

Сильно закрытые методы и атрибуты имеют **двойное подчеркивание** в начале их имен. Это приводит к искажению их имен, что означает, что к ним нельзя получить доступ извне класса.
Это делается не для того, чтобы гарантировать их конфиденциальность, а для того, чтобы избежать ошибок, если есть подклассы, у которых есть методы или атрибуты с одинаковыми именами.
Методы с измененным именем все еще доступны извне, но под другим именем. К методу `__privatemethod` класса `Spam` можно получить доступ извне с помощью `_Spam__privatemethod`.

```
class Spam:
    __egg = 7

    def print_egg(self):
        print(self.__egg)

s = Spam()
s.print_egg()
print(s._Spam__egg)
print(s.__egg)
```

```
7
7
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-13-98bece7fa29e> in <module>
      8 s.print_egg()
      9 print(s._Spam__egg)
---> 10 print(s.__egg)

AttributeError: 'Spam' object has no attribute '__egg'
```

По сути, Python защищает эти атрибуты, внутренне изменяя их имя - добавляя имя класса.

Статические методы и методы класса

Методы класса

Методы объектов, которые мы рассмотрели до сих пор, вызываются экземпляром класса, который затем передается в параметр `self` метода. **Методы класса** другие - они вызываются классом, который передается в параметр `cls` метода.
Обычно их используют фабричные методы, которые создают экземпляр класса, используя параметры, отличные от тех, которые обычно передаются конструктору класса.
Методы класса отмечены декоратором `classmethod`.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

    @classmethod
    def new_square(cls, side_length):
        return cls(side_length, side_length)

square = Rectangle.new_square(5)
print(square.calculate_area())
```

```
25
```

`new_square` - это метод класса, который вызывается в классе, а не в экземпляре класса. Он возвращает новый объект класса `cls`.

Технически параметры `self` и `cls` - всего лишь условные обозначения; их можно было поменять на что угодно. Тем не менее, они повсеместно соблюдаются, поэтому разумно их использовать.

Статические методы

Статические методы похожи на методы класса, за исключением того, что они не получают никаких дополнительных аргументов; они идентичны обычным функциям, принадлежащим классу.
Они отмечены декоратором `staticmethod`.

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @staticmethod
    def validate_topping(topping):
        if topping == 'pineapple':
            raise ValueError('No pineapples!')
        else:
            return True

ingredients = ['cheese', 'onions', 'salami']
if all(Pizza.validate_topping(i) for i in ingredients):
    pizza = Pizza(ingredients)
```

Статические методы ведут себя как обычные функции, за исключением того факта, что вы можете вызывать их из экземпляра класса.

Свойства класса

Свойства предоставляют способ настройки доступа к атрибутам экземпляра.
Они создаются путем помещения декоратора `property` над методом, что означает, что при обращении к атрибуту экземпляра с тем же именем, что и у метода, вместо этого будет вызываться метод.
Один из распространенных способов использования свойства - сделать атрибут доступным только для чтения.

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @property
    def pineapple_allowed(self):
        return False

pizza = Pizza(['cheese', 'tomato'])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
```

```
False
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-18-ee13d1018b2d> in <module>
      9 pizza = Pizza(['cheese', 'tomato'])
     10 print(pizza.pineapple_allowed)
---> 11 pizza.pineapple_allowed = True

AttributeError: can't set attribute
```

Свойства также могут быть установлены путем определения функций установки / получения.

Сеттер устанавливает значение соответствующего свойства.

Геттер получает значение.

Чтобы определить сеттер, вам нужно использовать декоратор с тем же именем, что и свойство, за которым следует точка и ключевое слово **setter**.

То же самое относится к определению геттера.

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings
        self._pineapple_allowed = False

    @property
    def pineapple_allowed(self):
        return self._pineapple_allowed

    @pineapple_allowed.setter
    def pineapple_allowed(self, value):
        if value:
            password = input('Enter the password: ')
            if password == 'Sw0rdf1sh!':
                self._pineapple_allowed = value
            else:
                raise ValueError('Alert! Intruder!')

pizza = Pizza(['cheese', 'tomato'])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
print(pizza.pineapple_allowed)
```

```
False
Enter the password: Sw0rdf1sh!
True
```
