Шпаргалка для Python

Бейбит Бактыгалиев, Sr Data Scientist,

Добро пожаловать!

Добро пожаловать на моё руководство по языку программирования Python. Этот справочник содержит важные темы, которые я обсуждаю на моем курсе “Python для начинающих” на YouTube.

Документ предоставляет краткое изложение материала из курса, позволяя вам ознакомиться с ключевыми концепциями и начать практиковать их на практике. Этот справочник будет идеальным дополнением к моему курсу и поможет вам быстро освоить язык Python.

О себе

Beibit Baktygaliyev
ucode.pro
beibit.ds
Beibit-ds

Я старший Data Scientist компании Амазон, одной из ведущих технологических компаний мира. У меня 19+ лет опыта программирования. Я начал с олимпиадного программирования, дальше работал в веб-разработке, базы данных и перешел в Data Science. В течение моей карьеры я имел честь сотрудничать с различными организациями, от крупных предприятий до стартапов, чтобы помочь им улучшить свои бизнесы с помощью технологий. Например, я был техническим лидером на проекте с Novartis AG, одной из ведущих фармацевтических компаний мира. Мы создали для них AI систему прогнозирования и поиск-рекомендации товаров что экономит им миллионы долларов в год. А если вы фанат Формулы-1, включите субтитры во время прямого эфира, и увидите работу моей команды. Мне также посчастливилось быть наставником, менторем, консультантом и помочь многим людям в сфере IT.

Contents

Variables (переменные)

В Python, переменная - это именованное место в памяти, которое хранит значение. Переменные можно объявлять, присваивая им значение с помощью оператора присваивания (=). Тип переменной определяется типом присваиваемого ей значения. Например, если переменной присваивается целочисленное значение, то она считается целочисленной переменной. Переменные могут быть переопределены новыми значениями в любое время, и их тип может измениться, если им присваивается значение другого типа.

```
name = "Beibit"  # string str
name = "Python"  # теперь переменная name смотрит на
строку "Python"
print(name)
age = 31    # integer int
pi = 3.14   # float
married = False  # Boolean bool
Married = True # married и Married разные переменные
```

Комментарии (Comments)

Комментарии в Python - это текст, который интерпретатор Python не интерпретирует и игнорирует. Они используются для добавления дополнительной информации и объяснения кода.

В Python есть два способа создания комментариев:

- Однострочный комментарий, который начинается с символа "#"

```
# Это однострочный комментарий
x = 5 # Это тоже однострочный комментарий
```

- Многострочный комментарий, который начинается с символа """ и заканчивается символом """

```
"""
Это многострочный комментарий.
Можно написать несколько строк текста
"""
```

Комментарии в коде могут помочь другим разработчикам понять, что делает код, и помогают вам понимать свой код позже. Они также могут использоваться для отключения кода без его удаления.

Считывание и вывод значений (Input и Output)

print() - это функция в Python, которая используется для вывода текста или других значений на экран. Она принимает один или несколько аргументов, которые должны быть выведены.

```
print("Hello, World!") # выводит строку "Hello,
World!"
x = 5
print("The value of x is", x) # выводит строку "The
value of x is 5"
```

input() - это функция в Python, которая используется для считывания ввода от пользователя. Она принимает один аргумент - строку, которая будет выведена как приглашение к вводу. Введенное значение возвращается в виде строки.

```
name = input("What is your name? ") # выводит строку
"What is your name?" и считывает ввод от
пользователя
print("Hello, " + name + "!") # выводит строку
"Hello, <введенное имя>!"
```

Обратите внимание что input() считывает строку. Если нужно использовать введенное значение как число например, нужно использовать типы int() или float().

Строки (Strings)

В Python, строковые данные представлены типом string (str). Строки могут быть заключены в кавычки (одинарные или двойные).

Создание строки

```
string1 = "Hello, World!"
string2 = 'Hello, World!'
```

Конкатенация строк

```
string3 = "Hello, " + "World!"
```

Дублирование строки

```
string4 = "Hello! " * 3 #  "Hello! Hello! Hello! "
```

Доступ к символам в строке и срезы

```
string1[0] # "H"
string1[-1] # "!"
string1[0:5] # "Hello"
```

Строки в Python являются неизменяемыми, то есть нельзя изменить отдельный символ в строке. Если нужно изменить строку, нужно создать новую строку с нужными изменениями.

Работа с методами строк

В Python строки имеют множество методов, которые можно использовать для работы с ними. Некоторые из самых популярных методов:

- upper() - преобразует все символы в строке в верхний регистр
- lower() - преобразует все символы в строке в нижний регистр
- count(substring) - возвращает количество вхождений подстроки substring в строку

- replace(old, new) - заменяет все вхождения строки old на строку new в строке

```
string1.upper() # "HELLO, WORLD!"
string1.lower() # "hello, world!"
string1.count("l") # 3
string1.replace("World", "Python") # "Hello,
Python!"
```

Есть много других методов и операций, которые можно использовать со строками в Python, но выше приведены основные и наиболее часто используемые.

Арифметические операции

В Python есть встроенные операторы для выполнения основных арифметических операций:

- + сложение

```
x = 3 + 4 # x = 7
```

- - вычитание

```
x = 5 - 2 # x = 3
```

- * умножение

```
x = 2 * 3 # x = 6
```

- / деление

```
x = 8 / 2 # x = 4.0
```

- % остаток от деления

```
x = 7 % 3 # x = 1
```

- // целочисленное деление

```
x = 8 // 3 # x = 2
```

Обратите внимание, что при делении с помощью оператора / возвращается дробное число, даже если деление является целым

числом. Если нужно получить целочисленное значение используйте оператор //.

- ** возведение в степень

```
x = 2 ** 3 # x = 8
```

В Python также можно использовать сокращенные арифметические операторы для изменения переменной в процессе выполнения операции.

```
x = 10
x += 5 # x = 15
x *= 2 # x = 30
```

Это аналогично x = x + 5 и x = x * 2 соответственно

Условие (if)

В Python, как и в других языках программирования, можно использовать условия для проверки значений переменных и выполнения кода в зависимости от результата этой проверки.

Основные операторы условного ветвления - if, elif и else.

```
x = 5
if x > 0:
    print("x положительное число")
elif x < 0:
    print("x отрицательное число")
else:
    print("x равно нулю")
```

if - это оператор, который используется для проверки условия. Если условие является истинным, то выполняется код в блоке if.

elif - это оператор, который используется для дополнительной проверки условия, если предыдущее условие было ложным.

else - это оператор, который используется для выполнения кода, если все предыдущие условия были ложными.

Сравнения

В Python есть стандартные операторы сравнения, которые можно использовать для сравнения двух значений. Они возвращают значение True или False:

== равенство

```
x = 5
y = 3
print(x == y) # False
```

!= неравенство

```
x = 5
y = 3
print(x != y) # True
```

> больше

```
x = 5
y = 3
print(x > y) # True
```

< меньше

```
x = 5
y = 3
print(x < y) # False
```

>= больше или равно

```
x = 5
y = 3
print(x >= y) # True
```

<= меньше или равно

```
x = 5
y = 3
print(x <= y) # False
```

Сравнивать можно не только числа, но и строки (string), boolean и другие типы данных. Обратите внимание, что сравнение строк производится в лексикографическом порядке.

```
string1 = "Hello"
```

```
string2 = "world"
print(string1 < string2) # True
```

Цикл While

Цикл while в Python используется для выполнения блока кода несколько раз, пока заданное условие истинно. Синтаксис цикла while выглядит следующим образом:

```
while условие:
    блок кода
```

Например, чтобы напечатать числа от 1 до 5:

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

В этом примере, переменная i инициализируется со значением 1, затем в теле цикла while проверяется условие i <= 5. Пока это условие истинно, тело цикла выполняется, печатая значение i и увеличивая i на 1. Когда i становится больше 5, цикл прекращается.

Важно обеспечить изменение условия в теле цикла, иначе он будет выполняться бесконечно.

Еще пример, напечатать числа от 10 до 0, используя цикл while:

```
x = 10
while x >= 0:
    print(x)
    x -= 1
```

Цикл For

Цикл for в Python используется для итерации по коллекции данных, такой как список, строка или диапазон чисел. Синтаксис цикла for выглядит следующим образом:

```
for переменная in коллекция:
    блок кода
```

Например, чтобы напечатать элементы списка:

```
fruits = ["яблоко", "банан", "апельсин"]
for fruit in fruits:
    print(fruit)
```

В этом примере, переменная fruit итерируется по списку fruits и на каждой итерации присваивается очередной элемент списка. На первой итерации fruit присваивается "яблоко", на второй - "банан", и т.д. Тело цикла выполняется для каждого элемента в списке, в данном случае печатая его.

Еще пример, напечатать числа от 0 до 9, используя цикл for:

```
for i in range(10):
    print(i)
```

В данном примере, функция range(10) возвращает диапазон чисел от 0 до 9, переме.

Список (List)

List в Python - это тип данных, представляющий собой упорядоченную коллекцию элементов. Элементы могут быть любого типа данных, включая другие списки.

Чтобы создать список, можно использовать квадратные скобки [] и разделитель “,” :

```
fruits = ["яблоко", "банан", "апельсин"]
```

Чтобы добавить элемент в список, можно использовать метод append():

```
fruits.append("мандарин")
```

Чтобы получить элемент из списка по индексу, используется квадратные скобки:

```
print(fruits[1]) # выведет "банан"
```

Чтобы изменить элемент в списке по индексу, можно использовать оператор присваивания:

```
fruits[1] = "груша"
```

Чтобы удалить элемент из списка по индексу, можно использовать метод pop():

```
fruits.pop(1)
```

Чтобы получить длину списка, можно использовать функцию len():

```
print(len(fruits)) # выведет 3 для списка fruits = ["яблоко", "банан", "апельсин"]
```

Кортежи (Tuples)

Tuple (кортеж) в Python - это тип данных, представляющий собой упорядоченную коллекцию элементов. Элементы могут быть любого типа данных, включая другие кортежи.

Чтобы создать кортеж, можно использовать круглые скобки () или не указывать скобки вовсе, разделяя элементы запятыми:

```
fruits = ("яблоко", "банан", "апельсин")
```

Кортежи являются неизменяемыми, то есть после создания, их нельзя изменить.

Чтобы получить элемент из кортежа по индексу, используется квадратные скобки:

```
print(fruits[1]) # выведет "банан"
```

Чтобы получить длину кортежа, можно использовать функцию len():

```
print(len(fruits)) # выведет 3
```

Кортежи и листы имеют некоторые общие характеристики, но также имеют ряд важных отличий.

- Неизменяемость: Кортежи являются неизменяемыми, то есть после создания их нельзя изменить. Листы же являются изменяемыми.
- Безопасность: Кортежи могут использоваться в качестве ключей в словарях, так как они являются неизменяемыми. Листы же не могут использоваться в качестве ключей, так как они являются изменяемыми.
- Производительность: Кортежи могут быть несколько быстрее и использовать меньше памяти, чем листы, особенно если вам нужно использовать большое количество элементов.

Таким образом, кортежи могут быть полезны в ситуациях, где необходимо использовать неизменяемые коллекции или когда нужна безопасность и эффективность. Например, кортежи могут использоваться для хранения координат, константных данных или для передачи нескольких значений из функции.

Словарь (Dictionary)

Dictionary (словарь) в Python - это тип данных, который представляет собой коллекцию пар "ключ-значение". Ключи должны быть уникальными и неизменяемыми (например, строки или числа), а значения могут быть любого типа данных.

Чтобы создать словарь, можно использовать фигурные скобки {} и пары "ключ: значение", разделенные запятыми:

```
person = {"имя": "Иван", "возраст": 30, "адрес":
"Москва"}
```

Чтобы добавить элемент в словарь, можно использовать оператор присваивания:

```
person["рост"] = 180
```

Чтобы получить значение по ключу, можно использовать квадратные скобки:

```
print(person["имя"]) # выведет "Иван"
```

Чтобы изменить значение по ключу, можно также использовать оператор присваивания:

```
person["возраст"] = 35
```

Чтобы удалить элемент из словаря, можно использовать метод del:

```
del person["адрес"]
```

Чтобы получить все ключи или значения из словаря, можно использовать методы keys() и values() соответственно:

```
print(person.keys()) # выведет ["имя", "возраст", "рост"]
print(person.values()) # выведет ["Иван", 35, 180]
```

Словари можно также итерировать с помощью цикла for.

```
for key, value in person.items():
    print(key, value)
```

Словари очень удобны для хранения и доступа к данным по ключу и используются в различных сферах программирования.

Множество (Set)

Set (множество) в Python - это тип данных, который представляет собой коллекцию уникальных элементов без определенного порядка. Элементы могут быть любого типа данных, но должны быть неизменяемыми.
Чтобы создать множество, можно использовать фигурные скобки {} или функцию set(), и передать в нее итерируемый объект:

```
fruits = {"яблоко", "банан", "апельсин"}
fruits = set(["яблоко", "банан", "апельсин"])
```

Множества поддерживают множество операций, как объединение, пересечение, разность и симметрическую разность.

```
fruits1 = set(["яблоко", "банан", "апельсин"])
fruits2 = set(["мандарин", "груша", "апельсин"])
print(fruits1.union(fruits2)) # выведет {"яблоко", "банан",
"апельсин", "мандарин", "груша"}
print(fruits1.intersection(fruits2)) # выведет {"апельсин"}
```

Чтобы добавить элемент в множество, можно использовать метод add():

```
fruits.add("виноград")
```

Чтобы удалить элемент из множества, можно использовать метод remove():

```
fruits.remove("банан")
```

Чтобы проверить, содержит ли множество определенный элемент, можно использовать оператор in:

```
if "яблоко" in fruits:
    print("Яблоко есть в множестве")
```

Множества можно использовать для удаления дубликатов из списка или другой коллекции, а также для решения задач, связанных с математическими множествами.

Функции (Functions)

Функции в Python - это блок кода, который может быть использован многократно для выполнения одной и той же задачи. Функции могут принимать аргументы (параметры) и возвращать значение.
Чтобы создать функцию, используется ключевое слово def, за которым следует имя функции и список параметров в круглых скобках:

```
def say_hello(name):
    print("Привет, " + name)
```

Чтобы вызвать функцию, нужно написать ее имя, за которым следует список аргументов в круглых скобках:

```
say_hello("Иван") # выведет "Привет, Иван"
```

Функция может возвращать значение с помощью ключевого слова return:

```
def square(x):
    return x * x

result = square(5)
print(result) # выведет 25
```

Функции могут иметь значения по умолчанию для аргументов, которые могут использоваться, если аргумент не был передан при вызове функции:

```
def repeat_message(message, times=1):
    for i in range(times):
        print(message)

repeat_message("Привет") # выведет "Привет"
repeat_message("Привет", 3) # выведет "Привет" 3 раза
```

Функции могут принимать произвольное количество аргументов с помощью *args и **kwargs

```
def print_args(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(key, value)

print_args(1, 2, 3, name='John', age=25)
```

Работа с ошибками (Try-except)

В Python, можно использовать конструкцию try-except для обработки исключений, которые могут возникнуть во время выполнения кода. Это

позволяет избежать аварийного завершения программы и вместо этого выполнять определенное действие при возникновении ошибки. Конструкция try-except состоит из блока try и одного или нескольких блоков except. Код, который может вызвать исключение, помещается в блок try, а блок except содержит код, который будет выполнен, если исключение возникнет.

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Нельзя делить на ноль!")
```

В данном примере, мы пытаемся выполнить деление на ноль, которое вызовет исключение ZeroDivisionError, которую мы ловим в блоке except. В случае возникновения этого исключения, код в блоке except будет выполнен и выведет сообщение "Нельзя делить на ноль!". Если исключение не возникает, то код в блоке except не выполняется.
Вы можете использовать несколько блоков except для обработки различных типов исключений:

```
try:
    x = int("hello")
except ValueError:
    print("Неверное значение!")
except TypeError:
    print("Неверный тип данных!")
```

Вы можете использовать ключевое слово finally для выполнения кода, который должен быть выполнен в любом случае, даже если исключение возникло:

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Нельзя делить на ноль!")
finally:
    print("Код в finally всегда выполняется.")
```

Вы можете использовать ключевое слово else для выполнения кода, если никакое исключение не было возбуждено в блоке try:

```
try:
    x = 5 / 2
except ZeroDivisionError:
    print("Нельзя делить на ноль!")
else:
    print(x)
```

В этом примере, код в блоке try выполняется без исключения, поэтому код в блоке else выполняется и выводит результат деления 5 на 2.
try-except конструкция очень полезна для обработки ошибок и исключений в вашем коде, и позволяет писать более стабильный и надежный код.

Классы (Classes)

В Python, классы используются для создания объектов и определения их методов и свойств. Они позволяют группировать данные и методы, связанные с ними, в единый контекст.
Чтобы создать класс, используется ключевое слово class, а затем имя класса. Внутри класса определяется метод __init__, который является конструктором класса и вызывается при создании объекта.

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def print_value(self):
        print(self.value)

obj = MyClass("Hello")
obj.print_value() # выведет "Hello"
```

В этом примере, мы создали класс MyClass с методом __init__, который принимает один аргумент value и присваивает его атрибуту класса value.

Также есть метод print_value(), который выводит значение атрибута. После создания экземпляра класса obj и вызова метода print_value(), выводится строка "Hello".
Классы являются основной концепцией объектно-ориентированного программирования и используются во многих языках программирования, включая Python. Они позволяют организовывать код и реализовывать различные абстракции, которые могут быть использованы в различных частях вашего кода. Они также помогают сделать код более читаемым и поддерживаемым. Классы в Python также поддерживают множественное наследование и могут быть использованы вместе с модулями и пакетами для создания больших и сложных проектов.

Наследование (Inheritance)

Наследование - это концепция в объектно-ориентированном программировании, которая позволяет создавать новый класс, который наследует методы и атрибуты существующего класса. Это позволяет избежать дублирования кода и упрощает его поддержку.
Например, есть класс "Автомобиль", который содержит методы движения и методы управления. Если мы хотим создать класс "Грузовик", который имеет те же методы, но имеет дополнительные методы для работы с грузом, мы можем создать класс "Грузовик", который наследует методы класса "Автомобиль".

```
class Automobile:
    def move(self):
        print("Moving")

    def steer(self):
        print("Steering")

class Truck(Automobile):
    def load_cargo(self):
        print("Loading cargo")
```

```
truck = Truck()
truck.move() # "Moving"
truck.steer() # "Steering"
truck.load_cargo() # "Loading cargo"
```

В этом примере класс Truck наследует методы move и steer из класса Automobile, и имеет дополнительный метод load_cargo. Это позволяет избежать дублирования кода и обеспечивает более структурированное и поддерживаемое программирование.
Наследование также позволяет переопределять или дополнять методы и атрибуты родительского класса в дочернем классе. Например, если мы хотим изменить поведение метода move в классе Truck, мы можем переопределить его в классе Truck:

```
class Truck(Automobile):
    def move(self):
        print("Moving slowly due to heavy cargo")
    def load_cargo(self):
        print("Loading cargo")
```

Наследование является одной из основных концепций объектно-ориентированного программирования и используется во многих языках программирования, включая Python.

Полиморфизм (Polymorphism)

Полиморфизм - это концепция в объектно-ориентированном программировании, которая позволяет использовать общий интерфейс для различных типов объектов. Это позволяет работать с различными типами объектов без их явной идентификации.
Например, есть класс Vehicle, который содержит метод move(). Если у нас есть классы Car и Bicycle, которые наследуют класс Vehicle, мы

можем использовать метод move() для каждого из них, не зная точно какой класс используется:

```
class Vehicle:
    def move(self):
        pass

class Car(Vehicle):
    def move(self):
        print("Moving on four wheels")

class Bicycle(Vehicle):
    def move(self):
        print("Moving on two wheels")

vehicles = [Car(), Bicycle()]

for vehicle in vehicles:
    vehicle.move()

#Output:
#Moving on four wheels
#Moving on two wheels
```

Как видно из примера, мы можем использовать объекты Car и Bicycle как если бы они были объектами типа Vehicle, так как они реализуют метод move(). Это позволяет нам использовать один интерфейс для работы с различными типами объектов, при этом не нужно специально определять тип каждого объекта.

Помимо этого в Python существуют специальные методы __str__, __add__, __eq__ и т.д. которые позволяют определить свой собственный механизм для работы с объектами класса. Это делает код более гибким и позволяет работать с объектами класса как с объектами встроенных типов.