22. Конспект

Конспект поможет вам вспомнить информацию, которая была в уроке, пригодится для дальнейшего обучения и работы.

Оператор if

Оператор **if** — оператор ветвления, или условный оператор; запускает код, если условие выполняется. Если условие не выполняется, то ничего не происходит.

Схематично конструкция с if записывается так:

```
if условное_выражение:
    инструкции
    инструкции
    ...
```

if — оператор, который вводит условие.

Двоеточие (:) после условного выражения сообщает, что дальше будет действие или набор последовательных действий при выполнении условия.

Инструкции — действие с каждым элементом.

Обратите внимание, что все указания после двоеточия идут с **отступом в 4 пробела**.

Простейшие условные выражения содержат булевы переменные:

```
is_server_on = True

if is_server_on:
    print("Сервер работает")
```

Переменные типа bool

Переменная типа bool (от boolean) — тип переменной, у которой два значения — «истина» или «ложь», True или False, «да» или «нет».

Например:

```
is_server_on = True         # Сервер либо включен, либо выключен
```

```
is_user_online = False      # Пользователь либо онлайн, либо нет
type(is_server_on)

>>> bool
```

Значение можно переключить, т. е. инвертировать, с помощью **not** — логического отрицания:

```
is_server_on = not True
print(is_server_on)
>>> False
is_user_online = not False
print(is_user_online)
>>> True
```

Операторы сравнения

Вместо булевых переменных в условных операторах можно использовать операторы сравнения. Они возвращают такие же булевы переменные, как мы только что создавали.

Поэкспериментируем со значениями:

```
a = 5
b = 10
```

Оператор «больше» (>):

```
if a > b:
    print("Первое число больше второго")
```

Оператор «меньше» (<):

```
if a < b:
    print("Первое число меньше второго ")
```

Оператор «больше или равно» (>=):

```
if a >= b:
    print("Первое число больше или равно второго")
```

Оператор «меньше или равно» (<=):

```
if a <= b:
    print("Первое число меньше или равно второго")
```

Оператор «равно» (==):

```
if a == b:
    print("Числа одинаковые")
```

Не путайте оператор «равно» (==) и оператор присваивания (=).

Оператор «не равно» (!=):

```
if a != b:
    print("Числа разные")
```

Оператор in проверяет, есть ли значение a в переменной b, используется для строк:

```
a = "ошки"
b = "мошки"
if a in b:
    print("ошки в мошках")
```

Равенство и неравенство также активно применяется для сравнения строк, а вот сравнение, какая строка больше, практически не используется.

Пример со строками:

```
a = "кошки"
b = "мошки"

if a == b:
    print("Слова одинаковые")

if a != b:
    print("Слова разные")
```

Оператор if-else

Оператор **if-else** — условный оператор, который проверяет, выполняется ли условие после **if**. Если условие выполняется, то совершается действие, если условие не выполняется, то срабатывает блок **else** («иначе»).

Схематично конструкция с **if-else** записывается так:

```
if условное_выражение:
    инструкции
    инструкции
else:
    инструкции
```

else — все исключения из условий if.

Обратите внимание, что все указания после двоеточия идут с **отступом в 4 пробела**.

Оператор elif

Оператор **if** — оператор, который вводит первое условие.

Оператор **elif** (сокращенно от else if) — оператор, который вводит последующие условия. В одной условной конструкции оператор **elif** может вводиться сколько угодно раз. Он проверяется в том случае, если **if** не сработал.

Пример:
```
if a > b:
    print("Больше")
elif a < b:
    print("Меньше")
elif a == b:
    print("Равно")
```

Если нужно проверить все исключения из условий **if/elif**, используется оператор **else**:
```
if a > b:
    print("Больше")
elif a == b:
    print("Равно")
else:
    print("Меньше")
```

Множественные сравнения

В Python внутри одного условия можно выполнять несколько сравнений.

Например, проверить, лежит ли число в диапазоне от 0 до 100:
```
number = 77  # Поэкспериментируйте со значением

if 0 <= number <= 100:
    print("число от 0 до 100")
```

Проверяем, полагается ли бесплатный проезд пассажиру:
```
age = 5

if 0 <= age <= 7:
    print("Полагается бесплатный проезд")
```

Проверяем, будет ли вода жидкостью при определенной температуре:
```
temp = 55

if 0 < temp < 100:
```

```
    print("Вода будет жидкостью")
```

Оператор and

Когда мы проверяем несколько условий, причем эти условия должны выполняться вместе и одновременно, то используем **оператор and** («и»).

Пример:

```
a = 0
b = 0

if a == 0 and b == 0: # Проверяются оба условия
    print("Переменные a и b обе равны 0")
else:
    print("Одна или обе переменные не равны 0")
```

Выведет:

```
Переменные a и b обе равны 0
```

Пример, когда хотя бы одна из переменных не равна 0:

```
a = 0
b = 1

if a == 0 and b == 0:
        print("Переменные a и b обе равны 0")
else:
        print("Одна или обе переменные не равны 0")
```

Выведет:

```
Одна или обе переменные не равны 0
```

Таблица истинности для and

Программа возвращает **True**, если оба условия — **True**.

Условие 1

Оператор

Условие 2

Результат

True

and

True

True

True

and

False

False

False

and

True

False

False

and

False

False

Оператор or

Когда мы проверяем несколько условий, причем нам достаточно, чтобы только одно из условий выполнялось, то используем **оператор or** («или»).

Пример:

```
a = 0
b = 0

if a == 0 or b == 0: # Проверяется одно из условий
        print("Одна переменная или обе равны 0")
else:
        print("Ни одна из переменных не равна 0")
```

Выведет:

```
Одна переменная или обе равны 0
```

Поменяем в программе значения:

```
a = 0
b = 1
```

```
if a == 0 or b == 0:
        print("Одна переменная или обе равны 0")
else:
        print("Ни одна из переменных не равна 0")
>>> Одна переменная или обе равны 0
```

Рассмотрим случай, когда обе переменные не равны нулю:

```
a = 1
b = 1

if a == 0 or b == 0:
        print("Одна переменная или обе равны 0")
else:
        print("Ни одна из переменных не равна 0")
>>> Ни одна из переменных не равна 0
```

Таблица истинности для оператора or

Программа возвращает **True**, если хотя бы одно из условий **True**.

Условие 1

Оператор

Условие 2

Результат

True

or

True

True

True

or

False

True

False

or

True

True

False

or

False

False

Работа с числами и математические операции

Инкремент и декремент

Инкремент — это сокращенная запись для *a = a + n*:

a += n

Например:
```
deposit = 1000
deposit += 100
print(deposit)
>>> 1100
```

Декремент — это сокращенная запись для *a = a − n*:

a −= n

Например:
```
deposit = 1000
deposit -= 100
print(deposit)
>>> 900
```

Инкремент и декремент работают только **с одной переменной**, поэтому называются **унарными** операторами.

Возведение в степень

Возведение в степень обозначается двумя звездочками (**):
`a ** b`
.

Примеры:

1.
2. `number = 10 ** 2`

```
print(number)

>>> 100
```

3.
```
4. number = 3
5. exp = 3
6. result = number ** exp

print(result)

>>> 27
```

Операции деления

В Python три разновидности операций деления. Они часто используются для вычисления общих делителей, кратных, проверки четности и прочего, поэтому будут полезны при решении практических заданий на алгоритмы.

A/B

Вычисляет частное и всегда возвращает число с точкой (float):

```
a = 5
b = 2
print(a/b)
>>> 2.5
```

Результатом будет float, даже если число поделится нацело.

```
a = 4
b = 2
print(a/b)
>>> 2.0
```

A // B

Выполняет целочисленное деление, отбрасывая остаток:

```
a = 5
b = 2
print(a // b)
>>> 2
```

Результат будет целочисленным (int), если a и b — оба числа int. Однако если одно из чисел float, результат будет тоже float.

A % B

Вычисляет остаток после деления a на b:

```
a = 5
b = 2
print(a % b)
>>> 1
a = 4
b = 2
print(a % b)
>>> 0
```

Результат будет целочисленным (int), если a и b — оба числа int. Однако если одно из чисел float, результат будет тоже float.

Операция «остаток от деления» используется:

1. Для проверки четности числа.

```
n = 12

if n % 2 == 0:
    print(f"{n} — это четное число")
else:
    print(f"{n} — это нечетное число")
```

Вернет:

```
12 — это четное число
```

7. Проверки делимости.

Т. е. проверяется, делится ли одно число нацело на другое:

```
a = 12
b = 4

if a % b == 0:
    print(f" {b} — это делитель {a}")
else:
    print(f" {b} — это НЕ делитель {a}")
```

Вернет:

```
4 — это делитель 12
```

Запомните, у оператора % нет ничего общего с вычислением процентов!

Как рассчитывать проценты

Приоритет операций и скобки

В выражениях Python выполняет операторы с более высоким уровнем приоритета первыми.

Рассмотрим два выражения:

- `4 + 4 * 2`

  — в этом случае сначала выполнится умножение, а затем сложение.

- `(4 + 4) * 2`

  — здесь порядок исполнения меняется из-за скобок: сначала выполняется операция в скобках, затем — умножение.

Полная таблица приоритета операторов представлена здесь:

https://pythonru.com

Минимум, максимум и среднее арифметическое

Для частых операций у Python есть специальные функции.

Функция **min()** вычисляет минимальное значение:

```
min(число, число, число, ...)
```

Например:

```
result = min(5, 4, 3, 2, 1)
print(result)
>>> 1
```

Функция **max()** вычисляет максимальное значение:

```
max(число, число, число, ...)
```

Например:

```
result = max(5, 4, 3, 2, 1)
print(result)
>>> 5
```

Чтобы посчитать среднюю зарплату или среднее время ожидания, вычислим **среднее арифметическое значение**. Для этого сложим все числа и поделим на их количество:

```
total = 5 + 4 + 3 + 2 + 1  # total — всего
average = total / 5 # average — среднее арифметическое
print(average)
>>> 3
```

True и False. Преобразование из типов bool

Преобразования в bool

Кроме очевидного преобразования к булевому типу значений 1 и 0, функция **bool()** используется для преобразования любых значений уже известных нам типов.

Например:
ВыражениеРезультатbool(99)Truebool(1.7)Truebool("hi")True

Любое «ненулевое» и «непустое» значение считается равным **True** при его преобразовании к bool, а «пустые» и «нулевые» значения приравниваются к **False**.

Например:

- целое число **0** — это **False**;
- дробное число **0.0** — это **False**;
- пустая строка **"" — это False;**
- неопределенное значение **None** — это **False**.

Также «пустым» считается пустой список **[ ]**, пустой кортеж **()**, пустой словарь **{ }**, пустое множество **set()**. Эти структуры данных мы будем обсуждать в следующих уроках.

Преобразования в int, float, str

Значения типа **bool** преобразуются в хорошо известные нам типы.

При преобразовании в числовые типы **True** превращается в **1**, **False** — в **0**, при превращении в строку **True** и **False** становятся **"True"** и **"False"**.

int(True)1int(False)0float(True)1.0float(False)0.0str(True)"True"str(False)"False"

Рассмотрим, как это можно использовать на практике.

У нас есть задача — посчитать количество рабочих дней. Для каждого дня есть переменная типа bool, где указано, был ли день рабочим. Чтобы получить верный ответ, мы складываем несколько переменных:

```
working_on_monday = True
working_on_tuesday = True
working_on_wednesday = False

working_days_total = int(working_on_monday)
                   + int(working_on_tuesday)
                   + int(working_on_wednesday)

print(working_days_total)
```

Выведет:

```
2
```

Приведение вместо преобразования

В примере выше мы преобразовали bool в int, но на самом деле нам даже не нужно делать это вручную.

При математических операциях Python сам превращает bool в нужный тип, это называется **приведением** (coersion), или **имплицитным преобразованием**.

Упростим пример:

```
working_on_monday = True
working_on_tuesday = True
working_on_wednesday = False

working_days_total = working_on_monday
                   + working_on_tuesday
                   + working_on_wednesday
```

```
print(working_days_total)
```

Выведет:
```
2
```

Рассмотрим еще одну задачу. За каждый положительный ответ теста добавляется по 3.5 балла к общему результату, правильность каждого ответа указана.

Для решения задачи мы можем умножить каждое значение на 3.5 и получить результат:
```
answer_1_correct = False
answer_2_correct = True
answer_3_correct = True
answer_4_correct = False

score = answer_1_correct * 3.5
      + answer_2_correct * 3.5
      + answer_3_correct * 3.5
      + answer_4_correct * 3.5
```

Выведет:
```
7
```

Правила PEP 8

Чтобы программы было удобно писать, проверять и читать, в Python существуют рекомендации по оформлению кода
PEP (Python Enhancement Proposal) — документ, в котором описаны новые фичи для Python и задокументированы аспекты Python для сообщества, например стиль, дизайн. Восьмой документ — **PEP 8** — самый популярный, и в нем содержатся правила оформления кода.

Рассмотрим некоторые из них:

1. Название переменных пишется в стиле snake_case.

snake_case (англ. змеиный_регистр) — стиль написания составных слов, где имена пишутся:

- в нижнем регистре,
- в одно слово,
- с подчеркиваниями вместо пробелов.

Пишем так:
```
user_name = "Alex"

user_salary = 100000

user_age = 33
```

Так нельзя:
```
UserName = "Alex"

user_Salary = 100000

user age = 33
```

2. Не используйте зарезервированные слова языка.

Если вы используете название оператора, то получите ошибку.

Вот неполный список слов, которые нельзя использовать:
```
False  True  None  and  with/as  break  class  continue

def  del  elif  else  finally  for  from  global  if

import  in  is  lambda  not  or  pass  raise  return

try  while
```

Более подробный список смотрите здесь: https://pythonworld.ru

3. Не используйте названия встроенных функций.

Встроенные имена уже объявлены в Python. Хотя вы можете их переписать — не делайте этого. Это порождает ошибки, о причинах которых не так просто догадаться новичку.

Например, при создании переменной **print** Python не выдаст ошибку, но сделает использование функции print невозможной.

Так можно:
```
greeting_text = "Добро пожаловать"

number_of_users = 4

max_guests_num = 50
```

Так нельзя:
```
print = "Добро пожаловать"

sum = 4

max = "Happy Birthday"
```

С некоторыми функциями вы уже знакомы:
```
bool()  float()  input()  int()  print()  round()
str()  type()
```

Список встроенных функций

4. Не используйте обманчивые символы.

Не используйте следующие символы как однобуквенные идентификаторы:

- l (маленькая латинская буква «эль») и I (заглавная латинская «ай»);
- 0 (ноль) и o (латинская буква «о»).

В некоторых шрифтах эти символы не отличимы от цифры 1 и 0. Если очень нужна l, пишите вместо нее заглавную L.

Пример плохого кода:
```
l = 100
I = 10
o = 50

result = l - 1 * o + 0.1 + I
```

5. Используйте пробелы вокруг операторов.

Чтобы визуально разделять название переменной, оператор и значение.

```
students_count = 12
group_id = 1620
teacher_name = "Дмитриева Клементина"
```

6. Ставьте пробелы после запятой при вызове функций.

Пример:

```
print ("оформление", "это", "важно")
```

7. В блочном коде ставьте четыре пробела.

Пример:

```
if pep == 8:
    print ("Всё отлично!")
```

8. Логические блоки разделяются пустыми строками.

Пример:

```
price = int (input ())

payment = price / 10
payment_round = round(payment)

print(payment_round)
```

9. Используйте пустую строку в конце кода, а точнее — символы завершения строки в конце файла — решетки (#).

Пример:

```
##############
if pep == 8:
    print("Всё отлично!")

##############
```

Добавляя пустую строку, мы видим более законченный формат кода, при этом интерпретатору проще его считывать.

Советы по именованию переменных

Теперь рассмотрим рекомендации, которых придерживаются разработчики. В соответствии с этими правилами мы будем рецензировать ваши работы.

1. Уделяйте внимание выбору имен для переменных.

Удачные имена для переменных ускоряют разработку, снижают количество ошибок и повышают вашу удовлетворенность работой.

Неудачный выбор может запутать вас уже во время написания программы, может замедлить вас, когда вы вернетесь к собственному коду, может запутать ваших коллег, которые будут работать с вами вместе.

2. Не используйте одно и то же имя для разных типов данных и разных значений

Хотя это экономично, но запутает вас и создаст проблемы в будущем.

Так хорошо:
```
money = 10000
days = 30

money_per_day = money / days
```

Так плохо:
```
money = 10000
days = 30

money = money / days
```

3. Избегайте переменных с названиями типа *a, v, x, y, i*.

По таким названиям невозможно предположить, о чём идет речь, какая задача выполняется, не нарушена ли логика и какой примерно тип у каждой переменной.

Чтобы дописать или исправить такой код, сперва нужно будет восстановить, что означает каждая переменная.

Так хорошо:

```
money_user_has = 10000
days = 30

money_per_day = money / day
```

Так плохо:

```
m = 10000
d = 30

y = m / d
```

Однако для таких названий есть **исключения**.

Первое исключение — использование букв в их традиционном значении:

- i — индекс (порядковый номер);
- v — скорость;
- x, y — координаты;
- r — радиус;
- a, b — обычно два числа в математических операциях.

Второе исключение — спортивное программирование или тренировка. Когда важна скорость, написание короткого кода — это необходимость.

Обратите внимание, что PEP 8 напрямую запрещает переменные **l** и **o**.

4. Избегайте переменных с названиями вроде *key, value, string, data* в тех случаях, когда название ничего не говорит об их содержании.

Если вы сомневаетесь, спросите коллегу, что лежит в переменной, которая так называется.

Так хорошо:

```
print("Введите имя и зарплату")

name = input()
```

```
salary = int(input())

print(f"{name} получает {salary}")
```

Так плохо:

```
print("Введите имя и зарплату")

data = input()
value = int(input())

print(f"{data} получает {value}")
```

5. Выбирайте существительные для названия переменных.

Хорошо, когда название переменной отвечает на вопрос «что здесь лежит?», а не «что с этим делать?».

Так хорошо:

```
number_of_boxes = 10

user_salary = 100 000

hours_to_sleep = 8
```

Так плохо:

```
how_much = 10

pay = 100 000

sleepy = 8
```

6. Для булевых переменных (bool) выбирайте имена, описывающие статус.

Например, **is_active**, **has_children**, **does_damage** и так далее.

Так хорошо:

```
has_children = True

is_alive = True

was_used = False
```

Так плохо:

```
children = True

live = True

use = False
```

7. Если вы понимаете, что название переменной недостаточно говорящее, то снабдите ее комментарием.

Пусть вашим коллегам будет проще разобраться. Особенно это верно для проектов со специальной терминологией.

Пример:

```
user_deposit_sum = 500000

payout_freq_in_days = 30    # Как часто делаются платежи (в днях)
interest_rate_percent = 6   # Процентная ставка
```

8. Читайте чужой код, анализируйте его с точки зрения именований.

Обменяйтесь работами с однокурсниками и заимствуйте имена для всего подряд, если они вам понравились.

9. Поменяйте имена переменных, даже если программа уже написана.

Если вы не можете придумать подходящее имя для переменной сразу, то выждите некоторое время, переберите несколько вариантов. Если хорошая идея пришла уже после того, как вы написали программу, то не ленитесь и перепишите ее.

Другой вариант — обратитесь к коллегам за советом. Если вам предложили более стройную систему имен, то обновите свои названия переменных, даже если программа уже работает.

Учите английский и активно пользуйтесь переводчиком — это позволит вам писать более прозрачные имена для переменных.

Константы

Константа — данные в программе, изменение которых после первого объявления запрещается напрямую (JavaScript, C) или не предполагается на уровне соглашения между разработчиками (Python).

В Python для обозначения переменных, значения которых не должны меняться во время работы программы, существует договоренность **именовать их прописными буквами**.

Отдельного синтаксиса для констант в Python нет, но константы упоминаются в PEP 8: https://peps.python.org.

В константах могут храниться данные любого типа:

```
# Стандартное количество баночек в упаковке
CANS_IN_PACKAGE = 32
# Приветствие пользователя
USER_GREETING = "Добро пожаловать в систему"
# Разрешение экрана
SCREEN_WIDTH = 640
SCREEN_HEIGHT = 480
# Путь к файлу
SOURCE_PATH = "/docs/source/data.txt"
# Путь к API
SOURCE_URL = "https://sky.pro/api/load-data/"
# Количество слов на странице
SYMBOLS_PER_PAGE = 1800
```

Константы обычно размещаются в самом начале программы.

В отличие от других языков программирования константы в Python не имеют специальной области видимости, ограничений на изменение и ведут себя в точности так, как и другие переменные.

Если константы — это обычные переменные, можно ли изменять константы в Python?

С точки зрения интерпретатора это не будет являться ошибкой, и такой код сработает:

```
# Задаем константу
CANS_IN_PACKAGE = 32
# Изменяем константу
CANS_IN_PACKAGE += 8
```

Но делать так не стоит — это осуждается сообществом разработчиков.