Лекция № 2

Т. Ф. Хирьянов

Преимущества языка программирования Python

1. *Современность.*
   Встроенно очень много удобных методик программирования.

2. *Универсальность.*
   На этом языке можно программировать любое приложение от скрипта операционной системы до игры на мобильном телефоне.

3. *Богатая стандартная библиотека.*
   В ней предусмотренно огромное количество функций, включая работу с сетями и математическими выражениями.

4. *Кроссплатформенность*
   Интерпретатор Python может работать в любой операционной системе, на компьютерах с разной архитектурой.

5. *Интерпретируемость*
   Одним из следствий интерпретируемости является то, что в переменную можно сохранять данные разных форматов.

   ```
   x = 123 x = "python"
   ```

   Так 123 – это целое число, а python – строка.

Ссылочная модель данных в Python. Динамическая типизация.

В Python нет операции присваивания. Запись

```
x = 123
```

означает, что объект 123 связывается с ссылкой x. А сама операция является связыванием объекта и ссылки. Кусок кода

```
x = int(x)
```

будет выполняться следующим образом. Сначала выполнится выражение стоящее справа, затем порожденный им объект, сохранится в некоторой области памяти, вообще говоря, отличающейся от того участка, на который указывал x ранее. В заключении ссылка x связывается с этим новым объектом.

Для того чтобы справиться с утечкой памяти старый объект удаляется сборщиком мусора, если на него больше нет ссылок.

При таком подходе тип объекта строго определен, но появляется у ссылки только в момент выполнения программы.

Отличия языков программирования Python 2 и Python 3

При развитии и улучшении языков программирования часто бывает необходимо кардинально изменить концепции привычных вещей. При таком переходе теряется совместимость старых и новых версий языка. Так произошло с Python2 и Python3.

Пример.

Функция input() в этих версиях языка ведет себя по разному. Так в Python2 выражение

```
x = input('5 + 3')
```

вычислит значение суммы и сохранит его в x. В Python3 это же выражение выведет на экран

```
5 + 3
```

а затем считает данные с клавиатуры и сохранит их в x в виде строки. Это бывает очень удобно. Например, можно вывести своеобразное приглашение: "Введите x".

Обмен двух переменных значениями

Допустим, есть две переменные

```
x = 3
y = 7
```

и необходимо произвести обмен их значениями. Эта задача однако не так тривиальна, как кажется. Например, такое решение

```
x = y
y = x
```

является неверным, так как после выполнения $x = y$ теряется ссылка на объект «3» и начинает указывать на «7».

В решении данной задачи удобно воспользоваться аналогией. Так, чтобы поменять содержимое стакана с водой и кружки с молоком друг между другом, можно воспользоваться третьим сосудом.

Этот алгоритм можно реализовать следующим образом.

```
tmp = x
x = y
y = x
```

Теперь нет потери значения x, так как оно предварительно сохранено в переменную tmp. Это классический пример обмена значениями переменных через третью.

Кортеж переменных

В языке Python существует элегантное решение этой задачи, основанное на использовании специальной структуры данных *кортежа переменных*.

Кортеж синтаксически представляет собой набор данных или переменных разделенных запятыми. Обычно он окружается круглыми скобками. Например,

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

является кортежем данных, а

```
(x, y, z)
```

кортежем переменных.

Такая структура имеет множество удобных способов применения. Так можно присвоить кортежу переменных кортеж данных.

```
(x, y, z) = (1, 2, 3)
```

Или обменять значениями две переменные более наглядным способом.

```
(x, y) = (y, x)
```

Более того, используя кортежи есть возможность организовать циклический сдвиг нескольких переменных.

```
(x, y, z) = (z, x, y)
```

Арифметические операции

Большинство арифметических операций синтаксически реализованы в Python нативным образом, как во многих других языках программирования. Например, операции сложения, умножения и деления выглядят следующим образом.

```
x + y
x * y
x / y
```

Однако существуют некоторые особенности. Первая из них связанна с операцией целочисленного деления div. Во многих языках программирования она будет выполнена следующим образом.

```
-17 div 10 = -1
```

Однако в соответствии с очевидным уравнением

```
x * 10 + r = -17
```

это приведет к тому, что соответствующий остаток будет отрицательным. Это противоречит принятому в математике положению, что остаток от деления по модулю — неотрицательное число. Для решения это проблемы в приведенном выше уравнении можно уменьшить x на единицу. Тогда целая часть и остаток от деления будут следующими.

```
-17 div 10 = -2
-17 % 10 = 3
```

Цикл с предусловием

Для организации циклического выполнения каких-либо действий в Python предусмотрена конструкция цикла с предусловием. Она начинается с зарезервированного слова *while*. Затем записывается условие, при котором будут выполняться действия, записанные в *тело цикла*. После условия необходимо поставить двоеточие и перейти на новую строчку. Перед каждой следующей строкой тела цикла делается отступ. Для того чтобы выйти из тела цикла необходимо и достаточно перейти на новую строчку и убрать относительный отступ.

При однократном выполнении или *итерации* цикла в начале будет проверено условие и если оно выполнено, то начнется последовательное исполнение команд тела цикла и переход к новой итерации по их окончании.

Существуют также *функции управления циклом*. Например, с помощью функции *break* можно преждевременно выйти из цикла. Как правило это бывает нужно, в случаях проверки дополнительного условия в момент выполнения итерации. Условная конструкция оформляется аналогичным образом: после зарезервированного слова *if* следует само условие, оканчивающееся двоеточием. В случае выполнения условия будут выполнены команды, записанные ниже через отступ.

Также бывает необходимо при выполнении какого-то условия завершить выполнение текущей итерации и перейти к следующей. Для этого используется функция *continue*.

В качестве примера оформления можно рассмотреть псевдокод, реализующий бег человека на стадионе.

```
while sunny:
    if trouble:
        break
    run 100 metres
    if difficult:
        continue
    run extremelly last 100 metres
```

При таком алгоритме действий человек будет бегать, пока на улице стоит хорошая погода. В случае форс-мажора, например, травмы, бег прекращается. Но если ничего не произошло, то спортсмен будет бежать трусцой 100 метров. Затем, если он устал, снова проверит погоду, оценит, все ли в порядке, пробежит еще 100 метров трусцой, т. е. перейдет к следующей итерации цикла. Если же человек полон сил, то последние 100 метров круга он пробежит так быстро, как сможет.

Примером настоящего кода может служить следующая программа.

```
x = int(input())
while x < 100:
    s += x
    if x % 7 == 0:
        break
    x += 5
else:
    print(x)
print(s)
```

Она выполняет подсчет суммы арифметической прогрессии с разностью, равной 5, начиная с введенного пользователем числа (которое приводится к целочисленному типу). Подсчет суммы производится пока члены прогрессии меньше 100. Строка

```
s += x
```

означает, что значение переменной s увеличивается на x. При этом если текущий просуммированный член прогрессии делится нацело на 7 (т.е. его остаток при делении на 7 равен 0), то осуществляется выход из цикла. Для сравнения равенства двух чисел используется оператор ==. В завершении программы на экран выводится значение суммы.

Для того чтобы по окончании работы цикла было понятно, завершился ли он нормально или преждевременно, существует специальное дополнение к конструкции цикла. После тела на уроне while записывается else и ставится двоеточие. Инструкции записанные ниже через отступ будут выполнены только когда будет нарушено условие в заголовке цикла, т.е. когда последний завершится нормально. В приведенном примере на экран выведется последнее значение x. Функция break осуществляет полный выход из цикла, а значит, «команды else» будут также не выполнены.

В условной конструкции также есть дополнение else. Синтаксически она выглядит так.

```
if условие:
    действие A
else:
    действие B
```

Генерация последовательностей.

Генерация арифметической прогрессии от нуля до вводимого пользователем N, не включая последнего, с разностью, равной 1.

```
N = int(input())
i = 0
while i < N:
    print(i)
    i += 1
```

Удобно использовать понятие *инварианта цикла*. Это некоторое утверждение, которое является верным в начале каждой итерации цикла. В данном примере оно может быть следующим: «В момент начала каждой итерации распечатаны все целые числа от 0 до N, не включая последнее».

Вложенные циклы.

Задача вывода на экран показаний часов с 5 до 12, когда минуты равны 10, 20, 30, 40 и 50.

```
hour = 5
while hour <= 12:
    minute = 10
    while minute <= 50:
        print(hour, minute, sep=':')
        minute += 10
    hour += 1
```

Чтобы не прописывать распечатку пяти показаний для каждого часа, логично использовать еще один, вложенный, цикл, в котором итератор будет пробегаться по значениям минут.

Стоит отметить, что функция break осуществляет выход только из цикла, в чьем теле она записана, т.е. с ее помощью нельзя выйти из вложенного цикла.

Функции

Для того чтобы программа была понятной любому программисту, в том числе и ее автору некоторое время спустя, важно давать функциям и переменным содержательные названия. Например, нельзя понять вне контекста результат работы функции f(a, b). Однако можно ожидать, что функция max(a, b) возвратит максимальное из двух чисел a и b.

Рассмотрим реализацию последней в качестве примера синтаксиса.

```
def max(a, b):
    if a > b:
        return a
    return b
```

Заголовок объявления функции начинается с зарезервированного слова *def*. Ввиду того, что Python является интерпретируемым языком, типы данных аргументов функции не указываются,

так как они определяются только в момент вызова функции. Поэтому приведенная программа будет одинаково хорошо работать как с целыми, так и с вещественными числами.

В данном примере не используется конструкция с else, так как функция return возвращает результат работы функции (значение переменной a или b) и завершает ее выполнение. Поэтому выполнится либо return a, либо return b.

Также функция имеет возможность возвращать кортеж. Примером может служить функция, осуществляющая обмен двух переменных значениями.

```
def swap(x,y):
    return y,x
```

Лекция № 3

Т. Ф. Хирьянов

Позиционные системы счисления

Система счисления — это механизм кодирования чисел. Сами числа существуют независимо от формы его записи. Например, тысяча в римской системе счисления записывается, как M, а в арабской – 1000. При этом эти записи обозначают одно и то же число. Самой простой системой счисления является *унарная*. В ней существует только один символ, который в случае числа A ставится подряд A раз.

Числа записываются с помощью цифр. Количество цифр соответствует названию системы. Например, в троичной СС их 3 (0, 1, 2), в двоичной – 2 (0, 1). Рассмотрим запись числа в десятичной системе счисления

$$12345_{10} = 10000 + 2000 + 300 + 40 + 5 = 1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 + 5 * 10^0$$

а также в двоичной

$$1010101_2 = 2^6 + 2^4 + 2^2 + 2^0 = 64 + 16 + 4 + 1 = 85$$

Стоит отметить, что в двоичной системе таблицы умножения и сложения выглядят очень просто.

$\times$	0	1
0	0	0
1	0	1

$+$	0	1
0	0	1
1	1	10_2

Также она является наиболее удобной для компьютера.

Для того чтобы выполнить обратную операцию воспользуемся представлением числа по схеме Горнера. Например,

$$12345_{10} = (((1 * 10 + 2) * 10 + 3) * 10 + 4) * 10 + 5$$

Из данного представления легко заметить, что последняя цифра числа есть остаток от деления на основание СС. Действительно,

$$12345 \ \% \ 10 == 5$$

Пусть теперь

$$y = 12345 \ div \ 10 == 1234$$

Тогда

$$y \ \% \ 10 == 4$$

Продолжая аналогично получим все цифры числа. Для расчетов на бумаге удобно пользоваться следующим представлением.

Данный алгоритм можно организовать в виде цикла. Для простоты можно написать программу, выводящую на экран цифры числа в обратном порядке, причем само число меняется по ходу выполнения программы.

```
x = int(input())
while x != 0:
    print(x % 10)
    x //= 10
```

Существуют родственные двоичной системы счисления. Это четверичная, восьмеричная, шестнадцатиричная и т.д.

Рассмотрим таблицу соответствий этих четырех систем.

«2»	«4»	«8»	«16»	«2»	«4»	«8»	«16»
0	0	0	0	1000	20	10	8
1	1	1	1	1001	21	11	9
10	2	2	2	1010	22	12	A
11	3	3	3	1011	23	13	B
100	10	4	4	1100	30	14	C
101	11	5	5	1101	31	15	D
110	12	6	6	1110	32	16	E
111	13	7	7	1111	33	17	F

Внимательно посмотрев на нее, можно заметить, что когда заканчивается разряд в шестнадцатиричной системе счисления, в двоичной тоже оказываются исчерпанными уже четыре разряда. Аналогично разряду в восьмеричной системе счисления соответствует три разряда в двоичной, а в четвертичной — два. Так, например, для перехода из шестнадцатиричной системы необходимо вместо каждой цифры подставить ее четырехразрядное представление в двоичной системе.

$$3DE80C_16 = 0011\,1101\,1110\,1000\,0000\,1100_2$$

Классификация числовых алгоритмов

1. *число* → *число*
   Такие алгоритмы получаются при определении значения функции от данного числа.

2. *число* → *последовательность*
   С помощью таких алгоритмов по заданным числам (некоторым параметрам) генерируется последовательность. Обычно она вычисляется либо по выражению для ее члена, либо при помощи рекуррентных соотношений.

3. *последовательность* → *число*
   В данных алгоритмах обычно реализованны различные подсчеты (например, количества элементов, их суммы или произведения), поиск чисел с заданными свойствами, проверка упорядоченности.

4. *последовательность* → *последовательность*
   Например, фильтрация.

В третьем пункте обработка последовательности включает цикл, в котором поочередно пробегаются по ее элементам. При этом в зависимости от задачи перед переходом к следующей итерации реализуются соответствующие «микроалгоритмы».

Так, при подсчете количества элементов n, до начала цикла полагается $n = 0$, а затем в теле цикла выполняется всего одна команда $n \mathrel{+}= 1$.

При подсчете суммы s для начальной, пустой последовательности полагается $s = 0$. Это, конечно, не обоснованно, однако уже после первой итерации сумма станет верной. При этом выполнится команда $s \mathrel{+}= x$.

То же самое касается произведения p. Только в этом случае вначале полагается $p = 1$, чтобы на первой итерации, содержащей команду p *= x значение произведения не обратилось в ноль и итоговый результат не был искажен.

При поиске происходит сравнение текущего элемента с некоторым образцом. Результатом этого сравнения является логическое значение (true или false), которое может быть приведено к целому числу (соответственно 1 или 0). В данном алгоритме используется специальная переменная – флаг (f). Вначале полагается $f = 0$. Затем в каждой итерации выполняется $f = int(x == x_0)$. Тем самым по завершении цикла f будет содержать количество элементов, равных x_0.

При реализации данных алгоритмов сама последовательность может быть введена двумя способами. В первом из них вначале программы может быть поданно число элементов последовательности. Тогда их считывание удобно организовать при помощи цикла *for*, считывая вначале каждой итерации текущий элемент. При подсчете суммы это может выглядеть, например, таким образом:

```
N = int(input())
s = 0
for i in range(N):
    x = float(input())
    s += x
print(s)
```

Когда же исходно неизвестно количество элементов, но известно, что признаком конца последовательности является, например, число 0, удобно организовать считывание элементов иначе, используя цикл *while*. При этом число 0 не является последним элементом (в случае подсчета произведения это привело бы к его обнулению). В данном случае ноль — это *терминальный признак*. Переход к телу цикла осуществляется, только если выполнено условие $x! = 0$, т. е. когда значение x нетерминальное. При этом необходимо считать первый элемент до начала цикла, а последующие в конце каждой итерации. Тот же самый подсчет суммы может выглядеть следующим образом:

```
s = 0
x = float(input())
while x != 0:
    s += x
    x = float(input())
print(s)
```

В данной программе ноль будет считан в последней итерации, однако не войдет в сумму, так как при последующей проверке условия цикл завершится.

Такой механизм используется, например, в языке Си, где признаком конца файла является специальный символ *EOF*.

Фильтрация, как правило, не реализуется отдельно, используется при выполнении другой задачи, например, при подсчете среднего арифметического только четных элементов последовательности. Для этого в теле цикла команды-обработчики заключаются внутри условной конструкции, проверяющей, является ли элемент четным числом. Например, так.

```
if x % 2 == 0:
s += 0
n += 1
print(s/n)
```

В качестве примера генерации рекурсивной последовательности можно рассмотреть выведение на экран N-ого числа Фибоначчи. В данной последовательности первые два элемента равны 1, а каждый последующий – сумме двух предыдущих.

1 1 2 3 5 8 13 21...

Из этого следует, что для корректной работы программы не требуется хранить в памяти компьютера все члены. Для то чтобы получить следующий элемент, достаточно помнить только последнии два. Вначале алгоритма вводятся две переменные $f1 = 0$, $f2 = 1$ и счетчик $n = 1$. Можно заметить, что при таких значениях элемент с номером n хранится в $f2$. Это утверждение должно быть верным на протяжении всего времени работы программы. Пока номер n меньше N, значение $f1$ меняется на $f2$, а $f2$ – на сумму $f1$ и $f2$. Последнее удобно организовать с помощью присваивания кортежей. В конце итерации, конечно, необходимо увеличить счетчик n.

```
N = int(input())
f1 = 0
f2 = 1
n = 1
while n < N:
    f1, f2 = f2, f1 + f2
    n += 1
print(f2)
```

Запись чисел в Python

В Python существует способ записи чисел в отличной от десятичной системе счисления. Например, введенное число number может быть записано в семеричной системе счисления, указав в качестве второго аргумента функции int() ее основание.

```
number = input()
x = int(number, 7)
```

При этом в памяти компьютера числа будут храниться всегда в двоичной системе счисления и только во время вывода они приобретут необходимый вид.

Существует также способ задать переменной значение в виде числа в двоичной, восьмеричной или шестнадцатеричной системе счисления. Для этого в начале числа необходимо записать специальный литерал: $0b$ для двоичной, $0o$ для восьмеричной и $0x$ для шестнадцатеричной.

```
x = 0b10110
x = 0o756
x = 0xB708
```

Однопроходные алгоритмы

В приведенных ранее алгоритмах количество операций линейно зависело от размера последовательности чисел, а объем требуемой памяти был конечным и не зависел этого размера.

Бывает, что в сложных задачах математика может помочь уменьшить количество вычислений.

Так, например, вычисление среднего квадратичного отклонения от средней величины на первый взгляд осуществляется в два прохода, и в памяти требуется хранить все числа.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2)}{N}}$$

Действительно, кажется, что сначала нужно подсчитать $\bar{x}$, а затем уже само отклонение. Однако если раскрыть квадрат и произвести суммирование, то можно получить известную формулу.

$$\sigma^2 = \frac{\sum_{i=1}^{N}({x_i}^2 - 2x_i\bar{x} + \bar{x_i}^2)}{N} =$$

$$= \frac{\sum_{i=1}^{N} {x_i}^2}{N} - 2\bar{x}^2 + \frac{\bar{x}^2}{N}N$$

Откуда

$$\sigma = \sqrt{\bar{x^2} - \bar{x}^2}$$

Из этой формулы следует, что вычислить отклонение можно за один проход, определяя среднее значение x и его средний квадрат.

Логические операции

В питоне существуют логические операции *and*, *or* и *not*. Они возвращают значения *True* или *False* (в качестве обозначений будут использоваться соответственно 1 и 0) и имеют следующие таблицы истинности.

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

A	not A
0	1
1	0

Алгебра логики

Составные высказывания

В логике Аристотеля существуют два состояния: *истина* и *ложь*, которые удобно обозначать соответственно как 1 и 0. Изначально рассматривалось содержание высказываний. Однако впоследствии утвердился подход, в котором существуют атомарные (простые) высказывания, не разбивающиеся на отдельные. Их истинность зависит от сторонних знаний, и в конкретном рассмотрении атомарное выражение принимается истинным либо ложным аксиоматически. Так например, выражение $1 + 1 = 1$ можно считать ложным с точки зрения привычной арифметики либо истинным, если рассматривать алгебру, где это возможно.

Предметом же рассмотрения алгебры логики являются *составные высказывания*. Для их образования из простых высказываний используются *логические операции*. Они выполняются в определенном порядке, соблюдая приоритетность, которая убывает следующим образом.

1. *не A* (логическое отрицание).
   Эта операция имеет самый большой приоритет и применяется к одному высказыванию. На письме отрицание A (не A) записывается одним из двух равноправных способов:

   $$\neg A \qquad \bar{A}$$

2. *A или B* (логическое умножение)
   Такая операция применяется к двум выражениям. Ее запись также имеет альтернативу.

   $$A \wedge B \qquad A \cdot B$$

3. *A или B* (логическое сложение)
   Записывается аналогично.

   $$A \vee B \qquad A + B$$

4. *импликация*
   Эта операция может быть выражена с помощью предшествующих трех, однако из-за частой встречаемости обозначается специальным образом.

   $$A \to B \qquad A \Rightarrow B$$

5. *эквиваленция*
   Является прямой и обратной импликацией.

   $$A \leftrightarrow B \qquad A \Leftrightarrow B$$

6. *либо* (исключающее или)
   Обозначается так.

   $$A \oplus B$$

По своей сути эти операторы являются логическими функциями и, как любые функции, отображают множество, называемое областью определения, на множество, являющееся областью значений. Так как область определения в данном случае состоит из двух элементов – правды (1) и лжи (0) – то можно сопоставить каждому элементу соответствующее значение функции (тоже 1

или 0). Получается что для одного переменного существует $2 * 2 = 4$ функций, а для двух аргументов – $(2 * 2) * 2 = 8$ функций. Удобно записать это в виде таблицы. Для введенных ранее операций они будут выглядеть следующим образом.

A	not A
0	1
1	0

A	B	$A \cdot B$	$A + B$	$A \oplus B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	0
1	1	1	1	0	1	1

Так $A \cdot B$ возвращает истину только когда оба аргумента истинны, а $A + B$ наоборот возвращает ложь только когда оба аргумента ложны. Исключающее или $A \oplus B$ похоже на обычное или, однако в отличие от него возвращает ложь в случае истинности обоих аргументов; значение данной операции равно правде только в том случае, если один из аргументов является правдой, а другой обязательно ложью. Эквиваленция верна, когда оба аргумента равны друг другу.

Особого пояснения заслуживает импликация (следствие). Очевидно, что если из истинного утверждения следует истинна, то рассуждения верны, т.е. истинны $(1 \Rightarrow 1 = 1)$. Также истинно рассуждение, если из ложного утверждения получается ложное. Например, $2 + 2 = 5$ ложно, откуда следует, что и $4 + 4 = 10$ тоже ложно. Может показаться противоречивым, но из ложного суждения может следовать истинное. Так, $2 + 2 = 5$ ложно. Утверждение $5 = 2 + 2$ также ложно и является справедливым следствием первого утверждения (так как$0 \Rightarrow 0 = 1$). Так как эти утверждения ложны одновременно, то их логическое произведение возвращает ложь$(0 \cdot 0 = 0)$. Однако из этих двух утверждений вместе следует, что 2+2 = 2+2, а это истинное выражение, т.е. $0 \Rightarrow 1 = 1$. Импликация возвращает ложь только в случае, когда из верного утверждения следует ложное. Действительно, если из утверждения $2 + 2 = 4$ получается, что, например, $3 + 3 = 5$, то рассуждения, очевидно, не верны $(1 \Rightarrow 0 = 0)$. Анализируя полученные значения для импликации, можно заметить, что из ложного утверждения может следовать как правда, так и ложь. Именно поэтому в системе аксиом не должно быть противоречивых утверждений, так как тогда следствия из них не всегда будут верны.

Нормальная дизъюнктивная форма

Из указанных функций некоторые выражаются через другие. Из-за этого возникает вопрос, какого количества из этих операций будет достаточно для описания произвольной логической функции любого количества аргументов. Теория показывает, что трех из них (не, и, или) достаточно для этого, нужно только привести функцию к *нормальной дизъюнктивной форме*, т.е. к логической сумме специально сконструированных произведений, равных истине только в одном из случаев, когда функция также истинна. Значит, таких произведений будет столько, сколько раз функция принимает значение 1 в ее таблице истинности.

A	B	C	f(A,B,C)	$\bar{A} \cdot \bar{B} \cdot C$	$\bar{A} \cdot B \cdot C$	$A \cdot B \cdot \bar{C}$
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	0
0	1	1	1	0	1	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	1
1	1	1	0	0	0	0

Тогда нормальная дизъюнктивная форма для приведенной функции будет равна следующей величине.

$$f(A, B, C) = \bar{A} \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot C + A \cdot B \cdot \bar{C}$$

Из нее видно, что логического отрицания, умножения и сложения достаточно, чтобы представить логическую функцию в виде такой формы.

Законы алгебры логики

Очевидные соотношения:

- $A + 1 = 1$
- $A + 0 = A$
- $A \cdot 1 = A$
- $A \cdot 0 = 0$

Простые законы:

- $A \cdot A = A \qquad A + A = A$(законы повторения)
- $A \cdot \bar{A} = 0 \qquad A + \bar{A} = 1$ (законы «третьего лишнего»)
- $A \cdot (A + B) = A \; A + A \cdot B = A$ (закон поглощения)
  Действительно, подставив вместо A 1, а затем 0, получаем с помощью очевидных соотношений верные утверждения.

Свойства математических операций.

- $A \cdot B = B \cdot A \qquad A + B = B + A$ (коммутативность)
- $A \cdot (B \cdot C) = (A \cdot B) \cdot C = A \cdot B \cdot C \qquad A + (B + C) = (A + B) + C = A + B + C$ (ассоциативность)
- $A \cdot (B + C) = A \cdot B + A \cdot C \qquad A + B \cdot C = (A + B) \cdot (A + C)$ (дистрибутивность)

Законы отрицания.

- $\bar{\bar{A}} = A$
- $\neg(A + B) = \bar{A} \cdot \bar{B}$
- $\neg(A \cdot B) = \bar{A} + \bar{B}$

Представления операторов.

- $A \Rightarrow B = \bar{A} + B$
- $A \Leftrightarrow B = A \cdot B + \bar{A} \cdot \bar{B}$
- $AB = \bar{A} \cdot B + A \cdot B$

Определение четверти координатной плоскости

Рассмотрим часто встречающуюся логическую задачу – определение множества, которому принадлежит элемент. В качестве множеств будут выступать координатные четверти плоскости, а элементом будет точка с координатами x и y. Заметим, что если ордината точки положительна, то точка может принадлежать первой или второй четверти. Чтобы выбрать между ними, нужно проверить знак абсциссы. Если она положительна, то точка из первой четверти, если отрицательна, то из второй.

```
x = int(input())
y = int(input())
if y > 0:
    if x > 0:
        print('1')
    else:
        print('2')
else:
    if x > 0:
        print('4')
    else:
        print('3')
```

Заметим, что приведенное выше решение неверно в случае принадлежности точки осям координат. Это пример того, что перед написанием кода важно предварительно определить какими могут быть входные данные, будет ли задача корректной для них, какие случаи необходимо рассмотреть, чтобы решение было исчерпывающим.

Каскадная условная конструкция

При решении многих задач возникает типичная ситуация: входное данное в большинстве случаев (например, когда оно отрицательно) правильно обрабатывается некоторым алгоритмом.

Если множества, которым может принадлежать переменная, являются вложенными и их можно однозначно разделить по значению некоторых параметров, то удобно использовать каскадную условную конструкцию else ... if, которая ввиду частого употребления получила в Python укороченный синтаксис: можно просто записать elif. В приведенном ниже примере в качестве вложенных множеств выступают $(x < 0)$, $(x \leqslant 0)$, $(x < 1)$, $(x \leqslant 1)$.

```
x = int(input())
if x < 0:
    print('negative')
elif x == 0:
    print('null')
elif x < 1:
    print('0 < x < 1')
elif x == 1:
    print('1')
else:    # x > 1
    print('x > 1')
```

Поиск максимального значения

Для поиска наибольшего элемента в некотором наборе нужно задать начальное значение временного максимума, а затем сравнивать с ним каждый элемент. При нахождении элемента большего временного максимума, последний обновляется. В качестве начального значения лучше выбрать значение одного из элементов.

```
x = int(input())
tmp_max = x
while x!=0:
    if x > tmp_max:     # tmp_max = max(x, tmp_max)
        tmp_max = x
    x = int(input())
```

Допустим, что требуется найти три наибольших числа. Тогда использовать нужно уже три временных максимума, а сравнение значений элементов с ними теперь происходит внутри каскадной конструкции. С помощью кортежей обновление максимумов можно сделать очень просто.

```
x = int(input())
# задание начальных значений m1 > m2 > m3
while x!=0:
    if x > m1:
        m1, m2, m3 = x, m1, m2
    elif x > m2:
        m2, m3 = x, m2
    elif x > m3:
        m3 = x
    x int(input())
```

Тест простоты числа

В приведенной ниже реализации используется тот факт, что делитель числа не может превосходить квадратный корень последнего.

```
def isPrime(x):
    "x is a prime"
    delitel = 2
    while delitel**2 < x:
        if x % delitel == 0:
            return False
        delitel += 1
    return True
```

Разложение числа на множители

Для решения данной задачи необходимо проверить равен ли остаток от деления выбранного числа на некоторый делитель. Если это так, то дальше на делимость можно исследовать частное, иначе нужно переходить на следующий возможный делитель.

```
x = int(input())
delitel = 2
while x > 1:
```

```
if x % delitel == 0:
    print (delitel)
    x //= delitel
else:
    delitel += 1
```

Лекция № 5

Т. Ф. Хирьянов

Работа со списками в Python 3

Кодировки символов

Американская стандартная система кодирования ASCII позволяла сопоставить распространенным печатным и непечатным символам соответствующие числовые коды. Изначально она включала в себя латинский алфавит. С помощью изменения старшего бита на 1 имелась возможность кодировать национальные символы. Из-за этого возникло очень много различных национальных кодировок. Более того, такой подход не позволял писать международные письма, использующие одновременно несколько кодировок.

Это привело к созданию международного стандарта Unicode, который предлагал несколько кодировок, среди которых UTF-16, UTF-32. Первая из них позволяла кодировать 65532 символа, а вторая приблизительно $4 \cdot 10^9$, причем в них на каждый символ отводилось одинаковое количество памяти. Из-за довольно большого размера таких равномерных кодировок была предложена новая UTF-8, которая являлась неравномерной, т.е. в ней разные символы имеют разную длину. Также это позволяет добавлять все новые и новые символы. Язык Python ориентирован на стандарт Unicode, поэтому в строках можно использовать любые символы, если знать их код.

Тип str

Для разных языков программирования существуют разные подходы к реализации символьного типа. Например, в Pascal это тип *char*, который является символьным. В языке программирования Си существует тип с таким же названием, который является числовым. В Python же не существует символьного типа, вместо этого есть тип «строка» (*str*). Чтобы обратиться к конкретному символу в строке необходимо указать его номер в качестве индекса.

```
s = 'Hello'
s[0] == 'H'
type(s[0]) == class<str>
```

Также символ можно понимать как срез строки длиной в один символ. При этом необходимо учитывать, что, например, в Pascal строки изменяемые. В Python это не так – для того чтобы изменить строку, необходимо породить новый объект и затем сделать привязку к старому идентификатору.

Функции строк

Для работы со строками одними из самых важных являются функции поиска подстроки в строке find и rfind, которые имеют похожий синтаксис.

```
s = 'Ehehe...'
s.find('he')   # return 1
s.rfind('he')  # return 3
```

При этом функция find возвращает позицию первого символа подстроки, найденной от начала строки. Но функция rfind осуществляет поиск с конца строки и, соответственно, возвращает позицию последнего элемента подстроки.

Для того чтобы найти количество вхождений подстроки в строку существует функция count. В случае перекрывания подстрок, как в данном примере,

```
s = 'AAAA'
num_AA = s.count('AA')    # num_AA == 2
```

можно считать, что подстрока вошла два или три раза. В Python 3 принято считать что здесь две подстроки.

Литералы строк

Строка в Python записывается в одинарных или двойных кавычках. Обе записи равнозначны и существуют для того, чтобы избегать двусмысленных выражений. Впрочем, можно было бы использовать обратный слэш для экранирования кавычек (например, \'), чтобы они воспринимались как символы внутри строки.

Если строка слишком длинная, чтобы уместиться на экране, то разумно перенести ее с помощью обратного слэша, который будет экранировать не печатающийся символ переноса каретки. При этом Python будет воспринимать ее как одну целую строку.

Для того чтобы работать с многострочным объектом, его записывают в тройных кавычках (каждая из которых состоит из одной либо двух кавычек).

```
s = 'Ehehe...'
s = "Ehehe..."

s_with_comma = 'I\'m a student'
s_with_comma = "I'm a student"
s_with_comma = 'Company "Parallels“

long_s = "...
..."

many_s = ''' ...
...
... '''
```

Для написания строк бывает удобно пользоваться специальными символами, такими как
«$\backslash n$» – разрыв строки
«$\backslash t$» – табуляция

Для того чтобы хранить в строке путь к некоторому файлу в Windows, необходимо экранировать обратный слэш (т.е. сделать его двойным). Можно также написать перед открывающей кавычкой символ r. Тогда все, что находится внутри кавычек будет восприниматься так, как написано.

```
path = 'C:\\my\\1.py'
path = r'C:\my\1.py'
```

Наивный поиск подстроки в строке

В качестве примера алгоритма можно рассмотреть поиск позиции первого элемента первого вхождения подстроки «he» в строку «Ehehe...».

Для начала определяются их длины. Затем элементы в строке пробегаются до той позиции, при которой подстрока, начинавшаяся бы с нее, еще не вышла бы за пределы строки. В теле цикла проверяется совпадение подстроки с частью строки, начинающейся с текущей позиции. В случае несовпадения происходит выход из вложенного цикла. Для того чтобы выйти из внешнего цикла сразу после нахождения нужной подстроки используется логическая переменная-флаг found. В случае, если такой подстроки нет, программа возвращает -1.

```
s = 'Ehehe...'
sub = 'he'
len_s = len(s)
len_sub= len(sub)

pos = 0
while pos + len_sub <= len_s:
    found = True
    i = 0
    while i < len_sub:
        if s[pos + i] != sub[i]:
            found = False
        i += 1
    if found:
        break
    pos += 1
else:
    pos = -1
```

Списки в Python

В Python список (тип list) представляет собой массив ссылок на объекты. Он изменяем: каждую ссылку можно связать с другим объектом.

```
A = [1, 2, 3, 4, 5]    # type(A) == class<list>
A[0] = 10              # type(A[0]) == int
                       # A == [10, 2, 3, 4, 5]
```

Можно узнать длину списка с помощью len(A), а также min(A) и max(A), если объекты, с которыми связаны ссылки можно сравнивать друг с другом.

Если присвоить один список другому, то тогда оба идентификатора станут указывать на один и тот же список.

```
A = [1, 2, 3]
B = A
B[0] = 10           # B == [10, 2, 3]
                    # A == [10, 2, 3]
```

Данная особенность связывания справедлива и при вызове функций. Так если в объявлении функции изменить сам объект, то при выходе из нее он останется измененным.

```
def f(B):
    B[0] = 10
f(A)
# B[0] == 10
```

Если же в объявлении функции изменяются только локальные списки, то ссылки на них теряются при выходе из функции.

```
def f(B):
    B = B[::-1]
f(A)
```

В данном примере идентификатор B связывается сначала со списком, с которым уже связан A. По этой ссылке создается новый объект – срез, и с ним связывается идентификатор B. Из-за этого ссылка на прежний объект теряется, и при выходе из функции список A остается неизменным.

Еще один пример показывает, как можно добавить элемент в конец списка и как объединить несколько списков в один, сохраняя тот же порядок элементов, что и в исходных.

```
A = [1, 2, 3]
B = A
B.append(4)         # B == [1, 2, 3, 4]
C = A + B           # C == [1, 2, 3, 4, 1, 2, 3, 4]
```

Копия списка

Создание копии списка можно проводить разными способами. Так, циклически пробежавшись по все элементам списка и добавив их в новый, получится копия прежнего. Однако то же самое можно сделать короче с использованием срезов.

```
B = []
for x in A:
    B.append(x)
# shorter
B = A[:]
```

Если необходимо обезопасить список от изменения после выполнения какой-либо функции, то разумно передавать ей не сам список, а его срез. Т.е. вместо f(A) писать f(A[:]).

Присваивание в срез списка

Присваивание в срез с двумя параметрами никогда не приводит к ошибке. В следующем примере элементы массива, начиная с позиции 1 до элемента с позицией 3, вырезаются, а на их место вставляется другой фрагмент.

```
A = [0, 1, 2, 3, 4]
A[1:3] = [10, 20, 30]    # A == [0, 10, 20, 30, 3, 4]
A[1:4] = []              # A == [0, 3, 4]
```

Присваивание же в срез с тремя параметрами может привести к ошибке, если размер присваиваемого списка не является подходящим (не укладывается в исходном списке с заданным шагом)

```
A = [0, 1, 2, 3, 4]
A[1:5:2] = [10, 20]      # A == [0, 10, 2, 20, 4]
A[1:5:2] = [10, 20, 30]  # Error!
```

Также имеется возможность вставки фрагмента между соседними элементами исходного списка.

```
A = [1, 2, 3]
A[1:1] = [5, 6] # [1, 5, 6, 2, 3]
```

Следующий пример показывает, как с помощью присваивания в срез можно создавать списки с довольно сложным порядком следования элементов.

```
# [1, 6, 2, 5, 3, 4]
A = [0]*6                 # A == [0, 0, 0, 0, 0, 0]
A[::2] = [1, 2, 3]
A[::-2] = [4, 5, 6]
```

Обращение списка

В некоторых случаях списки бывают достаточно большими и на их обращение с помощью создания среза может не хватить памяти.

```
A = list(map(int, input().split()))
i = 0
while i < len(A) // 2:
    tmp = A[i]
    A[i] = A[-1-i]
    A[-1-i] = tmp
```

Данный алгоритм прост. Для его понимания достаточно заметить, что отрицательный индекс означает перебор элементов списка от последнего (-1) к первому (-n) и что исходный список обходится только до половины, так как иначе новый список был бы обращен в прежний.

Циклический сдвиг

Зачастую в некоторых задачах требуется осуществить циклический сдвиг элементов в списке. Так, при сдвиге влево необходимо сначала сохранить первый элемент, чтобы потом, после работы цикла, поставить его на последнее место.

```
A = [0, 1, 2, 3, 4]
tmp = A[0]
for i in range(0, len(A) - 1):
    A[i] = A[i + 1]
A[-1] = tmp          # A == [1, 2, 3, 4, 0]
```

При наличии необходимой памяти, то же самое можно проделать с помощью срезов.

```
A[:] = A[1:] + A[0:1]
```

Кортежи

В отличие от списков *кортежи* являются неизменяемыми. В следующем примере показано, как объявить кортеж и как создать пустой. Важно отметить, что для устранения путаницы с переменными числового типа вводится специальная форма записи кортежа из одного элемента.

```
A = 1, 2, 3, 4, 5    # type(A) == class<tuple>
A = ()               # empty tuple
A = 1                # type(A) == int
A = (1, )            # type(A) == class<tuple>
```

Лекция № 6

Т. Ф. Хирьянов

Структурное программирование

Небольшие программы, имеющие порядка ста строк кода, обычно просты в понимании, однако чем больше она становится, тем сложнее уловить ее принцип работы, понять, что обозначает та или иная переменная, найти и исправить ошибку. Именно поэтому код программы должен быть структурирован.

Базовые принципы структурного программирования

- Программа состоит из
  1. *последовательного исполнения*
  2. *ветвлений*
  3. *циклов*
- Повторяющиеся участки кода оформляют в виде *функций*
- Разработка программы осуществляется пошагово *сверху-вниз*

В качестве иллюстрации можно рассмотреть следующую задачу. На вход поступают строки с автомобильным номерам и величиной скорости, с которой транспортное средство проезжает мимо поста ГИБДД.

Патрульный останавливает машины, превысившие скорость 60 км/ч, и озвучивает размер взятки, которая зависит от «привилегированности» номера: если в номере все три цифры разные, то 100 рублей, если есть две одинаковые, то 200, если три, то 1000. При этом, когда в конце рабочего дня приезжает начальник, его не уличают в превышении скорости, даже если это случилось. Программа должна вывести «зарплату» постового за день.

```
A238BE 73 ⟶ 100
B202CC 84 ⟶ 200
B555PH 71 ⟶ 1000
...
A999AA 100
```

Приведенный ниже код данной программы написан с соблюдением положений структурного программирования.

```
def solve_task():
    print(count_salary())

def count_salary():
    salary = 0
    licence_num, speed = input().split()
```

```
    while not chief(licence_num):
        if float(speed) > 60:
            salary += count_tax(licence_num)
        licence_num, speed = input().split()
    return salary

def chief(licence_num):
    return licence_num == "A999AA"

def count_tax(licence_num):
    return 0 # FIXME
```

Действительно, в нем код оформлен в функции, имеющие интуитивно понятные названия. Функция `solve_task` выводит на экран зарплату. При этом она не подсчитывает ее - это выполняет функция `count_salary()`. В последней используется переменная-счетчик `salary`, значение которой отражает текущий доход. В `licence_num` и `speed` записываются соответственно номер и скорость автомобиля. Пока номера считываются в цикле, происходит анализ того, не принадлежит ли текущий номер начальнику и какой размер взятки запросить. Это осуществляется посредством вызова функций `chief` и `count_tax`.

Важно отметить, что вместо вызова `chief` можно было явно прописать в условии `licence_num == "A999AA"`, однако от этого бы пострадала ясность и понятность кода.

На этапе разработки можно лишь объявить некоторые функции, запрограммировав их так, что бы они возвращали корректное, хотя и неверное, значение (как в случае с `count_tax`). Для того чтобы впоследствии вернуться к доработке программы удобно пометить необходимые места кода комментарием `FIXME`.

Прежде чем начинать программировать, нужно спроектировать программу: определить, как будет осуществляться взаимодействие между функциями(какие данные будут подаваться на вход и что будет возвращаться). Все это лучше всего прописать в текстовой документации к программе, однако предварительно можно просто указать в многострочном комментарии в начале тела функции. Например, таким образом.

```
def count_tax(licence_num):
    "2 numbers - 200
        3 numbers - 1000
        else - 100 "
    pass
```

Стек вызовов

Допустим, что существует программа A, в процессе выполнения которой вызывается функция B. В таком случае работа программы A должна быть приостановлена и начато выполнение функции B. При этом по окончании работы последней программа A должна возобновить свою работу с того места, где она была прервана. Для этого необходимо сохранить в стеке *адрес возврата*. Если теперь уже в процессе выполнения функции B была вызвана функция C, то выполняются аналогичные действия. При этом адрес возврата к B кладется «сверху» предыдущего.

После того, как выполнение функции C подойдет к концу, команда *return* обеспечит возврат к необходимому месту и удалит соответствующий адрес из стека. При этом переменные, которые завела функция C для своей работы являются локальными и по окончании ее работы также удаляются - это важно иметь в виду при проектировании программы.

Именованные параметры

Если при вызове функции один или несколько из передаваемых параметров в большинстве случаев имеют определенные значения, то удобно указать их явно в заголовке функции. Тогда они станут значениями по умолчанию и их можно будет не прописывать, при вызове функции. В приведенной ниже функции, меняющей пробел на другой разделитель `sep`, указано значение по умолчанию последнего (`sep='.'`).

```
def my_print(s, sep='.'):
    res = ''
    for symbol in s:
        res += symbol + sep
    print(res)

my_print('Hello')
```

Также именованные параметры при вызове можно менять местами. Однако для этого нужно указывать их явно.

```
def f(x, y):
    return x/y

f(1, 2)      # f(x=1, y=2)
f(y=1, x=2)
```

В Python существует много *итерируемых объектов*, по элементам которых можно пробегаться в цикле. Например, в приведенной ниже функции `x` пробегает все значения от нулевого до последнего элемента `A`, где `A` может являться как списком, так и строкой.

```
m = 0
for x in A:
    if x > m:
        m = x
```

В данном примере слово «Hello» будет выведено на экран побуквенно.

```
for symbol in 'Hello':
    print(symbol)
```

Генерация списков

В Python есть очень удобная возможность создавать списки с помощью обхода элементов итерируемого объекта. Для этого нужно в квадратных скобках указать следующее.
`new_A = [f(x) for x in A]`
где `f(x)` – значение, сопоставляемое каждому `x` из итерируемого объекта `A`.

В приведенном ниже примере список B будет содержать квадраты элементов списка `A`. Более того можно задать определенное условие, при котором элементы попадут в новый список, как при генерации `C` (попадут только четные).

```
A = [int(x) for x in input().split()]
B = [x**2 for x in A]
C = [x for x in A if x % 2 == 0]
```

В Python можно создавать и двумерные (и более) списки. В приведенном ниже примере в списке A окажутся два элемента – списки, содержащие по три элемента. Доступ к элементу и его изменение осуществляется с помощью индексов.

```
A = []
A.append([1, 2, 3])
A.append([4, 5, 6])
# A[1][2] == 6
A[1][2] == 7
```

Для списков существует операция повторения (*) с синтаксисом `x = [a]*N` (здесь `x` будет списком, в котором число a повторено `N` раз). При этом каждый элемент A будет ссылаться на объект «a». Если же создать список списков так, как показано в примере, то при изменении элемента одного из внутренних списков, соответствующая величина a в других также поменяется.

```
N = int(input())
M = int(input())
A = [0]*N
B = [[0]*N]*M
```

Чтобы избежать данной проблемы удобно использовать генератор вложенных списков, в котором в качестве значения элемента генерируемого списка выступает другой список тоже генерируемый. Например, таким образом можно создать таблицу умножения.

```
A = [[0]*N for i in range(M)]
A = [[i*j for i in range(10)]
     for j in range(10)]


A = [[0]*N for i in range(M)]
A = [[4*j + i for i in range(4)] for j in range(3)]
```

Или такого рода таблицу.

0	1	2	3
4	5	6	7
8	9	10	11

Полиморфизм в Python заключается в том, что типы параметров могут быть различны. Главное, чтобы с данными типами корректно работали команды тела функции.

```
def plus(a, b):
return a + b

plus (1, 2)
plus (1.5, 7.5)
plus ('ab', 'c')
```

По этой же самой причине список можно использовать вместо логического выражения: если список пуст - это будут интерпретироваться как ложь, иначе – правда.

```
A = []
x = input()
while x != '0':
    A.append(x)
    x = input()
while A:
    print(A.pop())
```

Лекция № 7

Т. Ф. Хирьянов

Поиск корней функции

Данная задача не является тривиальной. Например, для функции

$$f(x) = \sin \frac{1}{x}$$

В сколь угодно малой окрестности нуля найдется бесконечное колличество нулей. Более того, в прикладных задачах поведение функции неизвестно, а значит, между любыми двумя точками она может несколько раз пересечь ось абсцисс. Поэтому для решения данной задачи необходимо использовать выводы из математического анализа, а именно лемму о промежуточных значениях непрерывной функции. Из нее следует, что если функция непрерывна на отрезке $[a, b]$, то на нем она обязательно принимает все значения от $f(a)$ до $f(b)$ (при условии, что $f(b) > f(a)$). Поэтому если найден такой отрезок, на котором функция непрерывна и имеет противоположные по знаку значения, то на этом отрезке функция обязательно имеет корень.

Биссекция

Для описанной выше ситуации существует алгоритм, который позволяет сколь угодно точно определить корень функции. Идея алгоритма биссекции заключается в следующем. Вычисляется значение функции в середине отрезка. Если оно равно нулю, то корень найден, если больше либо меньше нуля, то производится сужение отрезка, содержащего корень. Действительно из постановки задачи следует, что $f(a)*f(b) < 0$. Следовательно либо $f(a)*f(c) < 0$, либо $f(c)*f(b) < 0$. Допустим, что верно первое. Тогда нужно заменить правую границу b отрезка $[a, b]$ на c. Получившийся отрезок $[a, c]$ вдвое меньше и удовлетворяет всем условиям задачи, а значит, с ним можно провести такие же действия. Из этого следует, что искомый корень после n таких итераций будет находится, например, в середине текущего отрезка с точностью до половины последнего. Задание необходимой точности обеспечивает выход из цикла.

```
f_a = f(a)
f_b = f(b)
while abs(b - a) > eps*2:
    c = (a + b)/2
    f_c = f(c)
    if f_c*f_a < 0:
        b = c                # a = a
    elif f_c*f_a > 0:
        a = c                # b = b
    else:
        break
```

Поиск в списке

Если производить поиск конкретного значения в неупорядоченном списке, то будет осуществляться последовательный перебор его элементов и количество операций будет сравнимо с N. Однако если список упорядочен по неубыванию, то можно воспользоваться аналогом алгоритма поиска корня функции методом деления пополам.

Сначала задаются значения индексов, ограничивающих индексы элементов списка. Затем в цикле диапазон значений индексов разделяют пополам аналогично предыдущему алгоритму (деление, конечно, целочисленное). Выход из цикла осуществляется, когда диапазон сокращается до одного элемента. При этом возвращается номер элемента массива наиболее близкого к искомому значению, но превышающего его. Если искомое значение меньше всех элементов, то возвратится ноль, если больше – N.

```
def upper_bond(key, A):
    A = sorted(A)
    l = -1
    r = len(A)
    while r > l + 1:
        m = (l + r)//2
        if A[m] > key:
            r = m
        else:                # A[m] ⩽ key
            l = m
    return r
```

Сортировка списка

Для того чтобы отсортировать список (сделать его упорядоченным) существует множество алгоритмов. Одной из самых долгих является сортировка обезьяны.

Сортировка обезьяны

Для реализации данного алгоритма необходима функция, перемешивающая элементы в массиве. В стандартной библиотеке Python есть такая функция shuffle. Ее можно подключить из модуля random следующим образом.

```
from random import shuffle
```

Описанная ниже функция `monkey_sort` перемешивает список, пока он не станет отсортированным.

```
def monkey_sort(A):
    while not is_sorted(A):
        shuffle(A)
```

При этом вызывается функция `is_sorted`, проверяющая, является ли текущий список упорядоченным.

```
def is_sorted(A):
    return A == sorted(A)
```

Так как количество перестановок n элементов равно $n!$, то количество операций пропорционально $n!$.

Сортировка вставками

Существует гораздо более быстрые алгоритмы. Одним из них является алгоритм сортировки вставками. В нем последовательно пробегаются все элементы, правее крайнего левого. Каждый следующий элемент вставляется в уже отсортированную часть списка, расположенную левее, причем так, чтобы упорядоченность сохранилась. В приведенном ниже примере использован циклический сдвиг, и соответствующая сложность алгоритма пропорциональна N^2. Однако алгоритм можно улучшить, если использовать рассмотренную ранее функцию upper_bond для поиска места, в которое необходимо переставить текущий элемент.

```
def insertion_sort(A):
    for i in range(1, len(A)):
        new_elem = A[i]
        j = i - 1
        while j >= 0 and A[j] > new_elem:
            A[j + 1] = A[j] # сдвиг
            j -= 1
        A[j + 1] = new_elem
```

Однако алгоритм можно улучшить, если использовать рассмотренную ранее функцию upper_bond для поиска места, в которое необходимо переставить текущий элемент.

Лекция № 8

Т. Ф. Хирьянов

Сортировка выбором

Рассмотренная ранее сортировка вставками может быть удобна, когда имеется необходимость сортировать некоторый массив элементов по мере его наполнения. В сортировке выбором на каждом этапе ищется тот элемент, который должен стоять н текущей позиции в отсортированном списке. Т.е. сперва находиться наименьший элемент и ставится в начало списка. Затем будет осуществляться поиск больших чисел; при этом они займут сразу правильные места и до конца сортировки переставляться не будут, в отличие от сортировки выбором.

Реализация данного алгоритма выглядит следующим образом. В цикле пробегаются все позиции кроме последней (когда n - 1 элементов будут отсортированы, очевидно, оставшийся элемент будет находиться на нужном месте). Для каждой позиции ищется наименьший элемент из всех оставшихся правее элементов, включая последний. Если среди них найдется число, меньшее элемента на текущей позиции, то они меняются местами. Таким образом к моменту выхода из внешнего цикла список будет отсортирован. Сложность данного алгоритма составляет $O(n^2)$.

```
for pos in range(N - 1):
    for i in range(pos + 1, N):
        if A[i] < A[pos]:
            tmp = A[i]
            A[i] = A[pos]
            A[pos] = tmp
```

Сортировка методом пузырька

Существуют структуры данных, в которых доступ до i-ого элемента происходит за не менее i шагов. В таком случае неудобно сравнивать далеко стоящие элементы. Можно осуществить проход списка, сравнивая соседние элементы и упорядочивая их в пределах каждой пары. По окончании прохода максимальный элемент окажется в крайней правой позиции, т.е. будет отсортирован. Применяя аналогичный алгоритм к оставшимся элементам, осуществляется сортировка всего списка.

Для реализации данного алгоритма заведем переменную prohod, показывающую, что на каждой итерации нужно пробежать N - prohod элементов, начиная с крайнего левого. Так как на первом проходе осуществляется N - 1 сравнений, то удобно организовать проходы с помощью цикла от 1 до N не включая последнего. Если какая-то сравниваемая пара неупорядоченна, то элементы в ней меняются местами. Как и в предыдущем примере сложность данного алгоритма составляет $O(n^2)$.

```
for prohod in range(1, N):
    for i in range(N - prohod):
        if A[i] > A[i + 1]:
            tmp = A[i]
            A[i] = A[i + 1]
            A[i + 1] = tmp
```

Сортировка дурака

В плохих случаях имеет асимптотику $O(N^3)$.

```
i = 0
while i < N - 1:
    if A[i] < A[i + 1]:
        i += 1
    else:
        tmp = A[i]
        A[i] = A[i - 1]
        A[i + 1] = tmp
        i = 0
```

Сортировка подсчетом

Допустим, что необходимо отсортировать следующие элементы.

1 2 3 2 2 0 7 6 5 2 3 8 8 1 1 2 3 0 9 8 8 7 6 3 2

С помощью метода пузырька или другими описанными ранее методами на решение данной задачи уйдет около $25^2 \gtrsim 600$ шагов. Однако можно заметить, что следующий массив содержит не больше 10 различных элементов, каждый из которых представляет небольшое неотрицательное число. Используя дополнительный массив счетчиков frequency можно понизить сложность сортировки до $O(N)$. Считывая в цикле элементы от 0 до 9, будем обновлять в массиве frequency частоту их вхождений. После этого останется только вывести эти элементы в порядке неубывания столько раз, сколько каждый элемент входил в исходный список.

```
frequency = [0]*10
digit = int(input())
while 0 <= digit <= 9:
    frequency[digit] += 1
    digit = int(input())
for digit in range(10):
    print(*[digit]*frequency[digit], end=' ')
```

Однако данный алгоритм будет работать очень долго, если среди возможных чисел будут очень большие. Например, пусть требуется упорядочить всего четыре числа.

7 1 512 1875514852

Используя сортировку подсчетом, потребуется обнулить более чем миллиард элементов массива frequency, на что уйдет приблизительно столько же операций. К тому же потребуется внушительный размер дополнительно используемой памяти.

Поразрядная сортировка

Данная сортировка применима только для целых чисел или коротких строк и имеет линейную асимптотику $O(N \cdot M)$, где M – характерный размер сортируемых чисел. Для больших чисел асимптотика ухудшается.

Допустим нужно упорядочить следующие числа.

753 58 236 200 18 211 214 758 812 7

Начнем сортировать числа по их младшим разрядам, т.е. по их последним цифрам. Тогда получим список, в котором числа, отличаются только последней цифрой, будут упорядоченны друг относительно друга (например, 211 и 214, 753 и 758).

200 211 812 753 214 236 7 58 18 758

Аналогично сортируя числа по всем остальным разрядам, получим упорядоченный список.

Для реализации данного алгоритма определим максимальную длину числа, входящего в список. Пробегаясь по всем разрядам до максимального (в десятичной системе счисления), произведем сортировку, аналогично тому, как показано выше.

```
A = [int(x) for x in input().split()]
max_num_len = max([len(str(x)) for x in A])
for radix in range(0, max_num_len):
    B = [[] for i in range(10)]
    for x in A:
        digit = (x // (10 ** radix)) % 10
        B[digit].append(x)
        A[:] = []
    for digit in range(10):
    A += B[digit]
```

Проверка типа входных данных

```
def sort_number(A):
    assert(type(A[0]) == 'class<int>')
```

Лекция № 9

Т. Ф. Хирьянов

Рекурсия

Когда при выполнении функции вызывается еще одна функция, происходит запоминание места, откуда совершается переход. После завершения работы этой «внутренней» функции осуществляется вовврат к этому месту. Поэтому нет никаких проблем для функции вызвать саму себя. Такой подход называется *рекурсией*.

При этом объекты, на которые ссылаются локальные переменные «вызывающей» функции f не совпадают с объектами, на которые ссылаются локальные переменные «вызванной» функции f. Тем самым, при вызове еще одной функции f происходит создание нового *пространства имен*, и благодаря этому сборщик мусора не удаляет объекты, созданные при предыдущих вызовах функции f.

Пример рекурсивного решения находит свое отражение в сказке о репе. У героя этой сказки деда возникла задача вытянуть репу из земли. Однако его усилий оказалось недостаточно, и он позвал на подмогу бабку. В двоем они также не смогли вытащить репу, и поэтому бабка позвала внучку. Аналогично внучка позвала собачку, собачка – кошку, кошка – мышку. Только после этого их совместных усилий хватило для того, чтобы вытащить репу. Данная сказка иллюстрирует один важный момент: чтобы рекурсия не была бесконечной необходимо, чтобы существовал *крайний случай* (в данном случае помощь мыши). При этом по мере спуска вглубь рекурсии задача упрощается (недостаток сил для того, чтобы вытащить репу, уменьшается). Данный уровень погружения называется *глубиной рекурсии*.

Создание матрешки

Хорошей иллюстрацией рекурсии может быть функция имитирующая создание матрешки `make_matroska`. В качестве аргументов в нее передаются размер матрешки `size` и количество матрешек `number`, которое требуется создать. Сначала создается (печатается) нижняя половина матрешки. Затем инициируется создание вложенной матрешки. Т.е. осуществляется рекурсивный вызов. Для того чтобы рекурсия не была бесконечной, необходимо в начале выполнения тела функции проверить, нужно ли еще создавать матрешки (`if number > 1`). Когда это условие не выполнится, выведется сообщение о создании наименьшей в данном случае матрешки. После этого будет осуществляться возврат к месту предыдущего вызова и завершаться выполнение тела функции – создаваться (печататься) верхняя половина матрешки.

```
def make_matroska(size, number):
    if number > 1:
        print('низ размера', size)
        make_matroska(size - 1, number - 1)
        print('верх размера', size)
    else:
        print('матрешечка размера', size)
```

При этом внутри тела функции можно выделить два участка: до рекурсивного вызова и после. Первая часть называется *прямым ходом рекурсии*, вторая – *обратным*.

Поиск НОД. Алгоритм Евклида

Рекурсия хорошо применяется при реализации алгоритма Евклида для поиска наибольшего общего делителя. Суть его заключается в том, что если a и b делятся на некоторое число, то и остаток от деления a на b также делится на это число. Благодаря этому можно переходить к исследованию делимости меньших чисел и рекурсивно применять к ним данный алгоритм, пока аналогичный остаток не станет равным нулю.

```
def my_gcd(a, b):
    if b == 0:
        return a
    else:
        return my_gcd(b, a%b)
```

Факториал

Примером неуместного использования рекурсии является реализация алгоритма нахождения факториала натурального числа, несмотря на то, что код выглядит очень лаконичным (особенно с использованием тернарного оператора).

```
def factor(n):
    return 1 if n == 0 else factor(n - 1)*n
```

Тем самым размер используемой памяти растет пропорционально количеству вызовов n. При больших n это может стать существенной проблемой.

Числа Фибоначчи

Другим «плохим» примером может служить рекурсивная функция вычисления числа Фибоначчи.

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

Несложно заметить, что данная функция вызывается избыточное количество раз. Действительно, `fib(n - 2)` будет определенно при еще вычислении `fib(n - 1)`. С ростом количества вызовов n количество операций будет увеличиваться пропорционально 2^n.

Быстрое возведение в степень

Операция возведения в степень может быть записана следующим образом.

$$a^n = \begin{cases} 1, \text{ при } n = 0 \\ a \cdot a^{n-1}, \text{ при } n \neq 0 \text{ и } n \text{ нечетное} \\ (a^2)^{n/2}, \text{ при } n \neq 0 \text{ и } n \text{ четное} \end{cases}$$

Последняя строчка дает возможность сократить количество этапов вдвое при рекурсивном вызове, если n – четное число. В случае, если n степень двух, количество операций пропорционально $\log_2 n$.

```
def fast_power(a, n):
    if n == 0:
        return 1
    elif n%2 == 1:
        return a*fast_power(a, n - 1)
    else:    # n%2 == 0
        return fast_power(a*a, n//2)
```

Ханойские башни

В качестве примера удачного применения рекурсии удобно рассмотреть задачу о Ханойских башнях. Ее суть заключается в том, что необходимо переложить пирамидку, составленную из блинов разной величины, на другой стержень. При этом за один ход можно перекладывать только один блин, причем так, что бы больший блин не лежал на меньшем, т.е. на каждом стержне находились «правильные» пирамидки.

В приведенном ниже решении пирамидку из n блинов перекладывают с i на j стержень. Крайним случаем является пирамидка из одного блина – тогда решение тривиально. Он совпадает со случаем, когда остался только один не переложенный блин. Если допустить, что существует решение этой задачи для пирамидки из $n-1$ звеньев, то легко найти решение и для необходимой пирамидки. Действительно, нужно переставить пирамидку из $n-1$ звеньев на дополнительный свободный стержень, оставшийся самый большой блин переложить на нужный стержень и затем переставить остальные звенья поверх него. При этом задача перекладывания меньшей пирамидки аналогична исходной, поэтому удобно для данного действия вызвать ту же функцию, изменив, конечно, параметры. В приведенном ниже примере кода «перекладывание» заключается в распечатывании соответствующей строки.

```
def hanoi(n, i=1, j=2):
    if n == 1:
        print('переставить 1', 'блин с', i, 'на', j, 'стержень')
    else:
        hanoi(n - 1, i, 6 - i - j)
        print('переставить', n, 'блин', n - 1, i, 'на', j, 'стержень')
        hanoi(n - 1, 6 - i - j, j)
```

Генерация комбинаторных объектов

Генерация двоичных чисел

Для генерации двоичных чисел удобно заполнять строчку нулями и единицами с помощью рекурсивного вызова и выводить двоичное число в крайнем для рекурсии случае. В приведенной ниже функции `bin_gen` на вход подаются количество цифр в числе `n` и строка `prefix`, в которую будет записываться часть генерируемого числа и которая будет передаваться каждой новой функции. Она необходима, так как число «собирается», удлиняясь при каждом новом вызове. Для обработки крайнего случая отдельно рассмотрена ситуация, когда n равно нулю. Если же это не так, то, удостоверившись, что n неотрицательно, осуществляется «сборка» числа по двум направлениям: в

одном случае цифра с номером n будет равна 1, во втором – 0. Очевидно, что рекурсия достигнет крайнего случая, и программа напечатает ровно 2^n чисел.

```
def bin_gen(n, prefix = ''):
    if n == 0:
        print(prefix)
    else:
        assert(n > 0)
        bin_gen(n - 1, prefix + '0')
        bin_gen(n - 1, prefix + '1')
```

Генерация перестановок в списке

Еще одним примером использования рекурсии может служить генерация всех возможных перестановок в списке. Для простоты пусть элементы списка – это целые числа от 1 до n + 1. Список с перестановкой будет «собираться» по аналогии с предыдущим примером. Когда длина списка станет равной n (крайний случай), элементы списка будут выведены на экран с помощью `print(*A)`. Для того чтобы сгенерировать все перестановки рекурсивный вызов осуществляется в цикле, в котором пробегаются необходимые элементы. Чтобы последние не повторялись используется проверка на то, входит ли текущий элемент в частично созданный список.

```
def perestanov_gen(n, A=[])
    if len(A) == n:
        print(*A)
    else:
        for x in range(1, n + 1):
            if x not in A:
                perestanov_gen(n, A + [x])
```

Стоит отметить, что количество вызовов такой функции будет крайне велико и важно не допустить достижения максимальной глубины рекурсии, разрешенной в Python.

Лекция № 10

Т. Ф. Хирьянов

Динамическое программирование

У рекурсивной функции вычисления числа Фибоначчи очень большая асимптотическая сложность (2^n). При этом для человека это не слишком сложная задача при наличии листка бумаги и ручки. Это отличие обусловлено тем, что человек, начинает подсчет не с числа с номером n, а с первого числа и идет по порядку. Тем самым он идет от простой задачи к сложной, а не наоборот, как реализованно в рекурсивном алгоритме. В этом заключается подход динамического программирования. В качестве примера рассмотрим решение той же задачи с его помощью.

Функция `fib`, получающая на вход номер числа Фибоначчи, который необходимо вывести, создает список необходимого размера, начинающийся с 0 и 1. Затем в цикле находятся последовательно все числа Фибоначчи, начиная со второго и заканчивая n. Для этого используется очевидная рекуррентная формула `F[i] = F[i - 1] + F[i - 2]`.

```
def fib(n):
    F = [0, 1] + [0]*(n - 1)
    for i in range(2, n + 1):
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

Может показаться, что данное решение значительно проигрывает рекурсивному по объему используемой памяти, однако это не так, потому что при каждом вызове рекурсивной функции на стек кладется текущий номер числа и адрес возврата. Поэтому количество необходимой памяти в процессе работы программы становится пропорционально $2n$. В случае динамического программирования памяти требуется меньше. Можно даже улучшить программу, запоминая только последние два числа. Время выполнения программы также пропорционально n.

Кузнечик

Допустим, что кузнечик прыгает от нулевой клетки до клетки с номером n. При этом за один прыжок, он может преодолеть одну или две клетки. Необходимо посчитать количество возможных траекторий. Допустим, что в клетку с номером $i-2$ можно попасть N_{i-2} способами, а в клетку с номером $i-1$ – N_{i-1} способами, тогда очевидно, что количество траекторий до i-й клетки равно $N_{i-1}+N_{i-2}$. Используя этот факт и решение задачи о нахождении n-ого числа Фибоначчи, получим требуемый алгоритм, в котором вычисляются количества траекторий до каждой i-й клетки и хранятся последовательно в списке K.

```
K = [0, 1] + [0]*(n - 1)
for i in range(2, n + 1):
        K[i] = K[i - 1] + K[i - 2]
print(K[n])
```

По своей сути это очень похожие задачи.

Количество траекторий с запрещенными клетками

Можно усложнить предыдущую задачу, запретив кузнечику прыгать на некоторые клетки. Для этого нужно создать специальный список запрещенных клеток, например, присвоив i-й клетке значение 0, если в нее можно прыгнуть и 1, если это запрещено. Далее, как и в предыдущей программе, создается массив K, в котором для каждой клетки хранится количество траекторий до нее. Отличие состоит лишь в том, что если i-я клетка запрещенная, то количество траекторий до нее равно нулю.

```
Denied = [0, 0, 1, 0, 0, 1, 0, 0, ... 1, 0, 0]
n = int(input())
K =[0, 1] + [None]*(n - 1)
for i in range(2, n + 1):
    if Denied[i]:
        K[i] = 0
    else:
        K[i] = K[i - 1] + K[i - 2]
```

Задача о минимальной стоимости

Допустим условно, что за нахождение в каждой клетке взимается определенная сумма денег. Необходимо определить минимальную сумму, которую требуется заплатить, чтобы добраться до нужной клетки. Для этого по аналогии с предыдущим примером задается список цен нахождения в каждой клетке Price, и список C, в который для каждой клетки будет задаваться минимальная стоимость ее достижения. Очевидно, что стоимость достижения i-й клетки будет равна сумме стоимости нахождения в ней и минимальной из стоимостей достижения соседних клеток.

В данной задаче возникает вопрос, какая именно траектория обладает наименьшей стоимостью. Для этого можно заполнять массив Path номерами клеток, через которые проходила искомая траектория. При этом нужно двигаться с конца, каждый раз выбирая ту клетку из двух, стоимость прибывания в которой меньше. Полученный массив Path в конце работы программы следует перевернуть.

```
Price = [10, 20, 5, 3, ...]
n = int(input())
C = [0]*(n + 1)
C[0] = Price[0]
C[1] = C[0] + Price[1]
for i in range(2, n + 1):
    C[i] = Price[i] + min(C[i - 1], C[i - 2])
print(C[n])

Path = [n]
while Path[-1] != 0:
    i = Path[-1]
    if C[i - 1] < C[i - 2]:
        Path.append(i - 1)
    else:
        Path.append(i - 2)
Path = Path[::-1]
```

Исполнитель король

Похожим исполнителем является *король*, который может шагать по шахматной доске вниз, вправо и по диагонали (однако при этом он не может ходить назад). Фигура начинает шагать из верхнего левого угла. Можно поставить аналогичную задачу нахождения кратчайшей траектории до конкретной клетки. Для этого необходимо сначала обработать клетки на верхней и левой границах. Создадим двумерный список K, размера $n \times m$ заполненный нулями. Затем с помощью двух циклов укажем количество траекторий до крайних клеток. Очевидно, что в условиях данной задачи оно будет равно 1. Для остальных клеток очевидно существует больше траекторий. Действительно до каждой из них можно добраться из трех соседних клеток: сверху, слева или по диагонали. Поэтому количество траекторий до клетки в i-ой строке и в j-ом столбце равно сумме количеств траекторий до каждой из тех трех соседних клеток. Пробежавшись по всем оставшимся клеткам в двойном цикле, заполним список K необходимыми значениями. В конце можно вывести количество траекторий до любой клетки, например, для клетки в нижнем правом углу.

```
m = int(input())
r = int(input())
K = [[0]*m for i in range(n)]
for i in range(n):
    K[i][0] = 1
for j in range(m):
    K[0][j] = 1
for i in range(1, n):
    for j in range(1, m):
        K[i][j] = K[i - 1][j - 1] + K[i - 1][j] + K[i][j - 1]
print(K[n - 1][m - 1])
```

Наибольшая общая подпоследовательность

Допустим имеются два числовых списка A и B одинаковой длины. Требуется найти и вывести F_{ij} – длину наибольшей общей подпоследовательности срезов A[:i+1] и B[:j+1]. Когда эта величина уже посчитана для индексов i-1 и j-1, остается сравнить A_i и B_j. Если они равны, то очевидно общая подпоследовательность станет больше на единицу. Иначе будет выбрана наибольшая из $F[i-1][j]$ и $F[i][j-1]$.

```
n = len(A)
m = len(B)
F = [[0]*(m + 1) for i in range(n + 1)]
for i in range(1, n + 1):
    for j in range(1, m + 1):
        if A[i - 1] == B[j - 1]: символ A и j символ B
            F[i][j] = F[i - 1][j - 1] + 1
        else:
            F[i][j] = max(F[i - 1][j], F[i][j - 1])
print(F[n][m])
```

Наибольшая возрастающая подпоследовательность

Допустим, что существует список A, и необходимо узнать F_i – длину наибольшей возрастающей подпоследовательности до i-ого элемента включительно. Поэтому $F[0] = 0$. Будем последовательно заполнять список F. При этом если для элемента i списка A найдется меньший его элемент j среди

элементов с номерами, меньшими i, и при этом $F[j] > F[i]$, то нужно обновить значение $F[i]$. После этого нужно будет включить элемент $[i]$ в эту подпоследовательность, а значит, увеличить $F[i]$ на единицу.

```
F = [0]*len(A)
for i in range(len(A)):
    for j in range(i):
        if A[j] < A[i] \
            and F[j] > F[i]:
            F[i] = F[j]
    F[i] += 1
print(max(F))
```

Лекция № 11

Т. Ф. Хирьянов

Быстрая сортировка Хоара

Допустим, что необходимо упорядочить солдат в строю по их росту. Решение данной задачи с помощью быстрой сортировки Хоара начинается с того, что выбирается произвольный солдат и весь строй разделяется на три группы по отношению к нему: тех, кто ниже ростом, тех, кто одинакового с ним роста и тех, кто выше. Получается, что люди во второй группе, уже отсортированы внутри нее. К тому же первая вторая и третья группы упорядоченны между собой. Однако внутри первой и третьей группы люди могут быть неупорядоченны. Осталось рекурсивно применить к этим группам тот же самый алгоритм.

Из сказанного следует, что данная рекурсивная сортировка относится к алгоритмам типа «разделяй и влавствуй», а также то, что массив данных сортируется на прямом ходу рекурсии. Подробный анализ показывает, что приведенный метод в худшем случае имеет асимптотику $O(N^2)$, однако вероятностно, в некотором «среднем случае» $O(N \cdot \log_2 N)$.

Рассмотрим для наглядности реализацию, в которой для этих трех групп создаются новые списки (хотя можно обойтись и без этого). Сначала обрабатывается крайний случай рекурсии, который заключается в том, что список из не более одного элемента уже отсортирован. Затем с помощью библиотечной функции `choice` выбираем произвольный элемент, который будет выполнять роль барьера. Сравнивая с ним остальные элементы списка A, создаем с помощью генераторов три новых списка `left, middle, right`. К первому и третьему рекурсивно применяем тот же алгоритм. В конце работы программы возвращаем конкатинированный список.

```
from random import choice
def hoar_sort(A):
    if len(A) <= 1:
        return A
    barrier = choice(A)
    left = [x for x in A if x < barrier]
    middle = [x for x in A x == barrier]
    right = [x for x in A x > barrier]
    left = hoar_sort(left)
    right = hoar_sort(right)
    return left + middle + right
```

Стоит отметить, что важным преимуществом быстрой сортировки является ее универсальность, т.е. возможность ее применения к любым элементам, которые можно сравнивать между собой, даже если к ним не применимы арифметические операции, например, к строкам.

Быстрая сортировка слиянием

Представим, что студент распечатал внушительный отчет о проделанной лабораторной работе, однако из-за того, что форточка в комнате была открыта, ветер раскидал листки по комнате и

перепутал их. Предположим, что двое соседей студента решили ему помочь и каждый, собрав произвольную стопку листов, упорядочил страницы в ней. Эти две отсортированные стопки они отдали студенту. Теперь задача студента сводится к тому, чтобы произвести «слияние» этих двух стопок. Он ставит их так, чтобы страницы с наименьшими номерами были в верху стоп. Таким образом студент все время видит две страницы, выбирает из них ту, у которой номер меньше и откладывает в третью стопу.

Тем самым обработан крайний случай, когда уже имеется две отсортированных части списка. Последние можно получить рекурсивно. Для этого нужно, чтобы для каждой из них также существовали их отсортированные части. Легко заметить, что сортировка происходит на обратном ходу рекурсии. Асимптотика такого алгоритма равна $O(N \cdot \log_2 N)$, однако он потребляет $O(N)$ дополнительной памяти.

Для реализации данного алгоритма потребуется сперва создать функцию слияния `merge`, которая по двум отсортированным спискам A и B возвращает их общий упорядоченный список Res. По аналогии с описанным выше методом поэлементно пробегаются оба списка, пока каждый из них не закончится.

```
def merge(A, B):
    Res = []
    i = 0
    j = 0
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            Res.append(A[i])
            i += 1
        else:
            Res.append(B[j])
            j += 1
    Res += A[i:] + B[j:]     # один из срезов обязательно пуст
    return Res
```

Теперь реализуем функцию `merge_sort(A)`. Крайним будет случай, когда список A состоит из не более одного элемента. Тогда можно считать, что список уже отсортирован, однако если в нем больше элементов, то левую «половину» сохраняем в список `left`, а правую в список `right`. Далее для них выполняем те же самые действия рекурсивно. Стоит обратить внимание, что сортировка осуществляется на обратном ходу рекурсии, когда списки `left` и `right` подвергаются слиянию.

```
def merge_sort(A):
    if len(A) <= 1:
        return A
    left = A[:len(A) // 2]
    right = A[len(A) // 2:]
    left = merge_sort(left)
    right = merge_sort(right)
    return merge(left, right)
```

Пирамидальная сортировка

Данная сортировка использует специальную структуру данных «пирамида» (или «куча»), в которой самое большое число оказывается всегда на ее вершине. При добавлении нового числа в кучу, оно оказывается в самом низу пирамиды и затем поднимается по ступеням вверх, меняясь местами

с меньшими числами, расположенными выше, пока не достигнут необходимой. Затем когда пирамида заполнена (с помощью библиотечной функции `heapify`), числа начинают извлекать из нее (`heappop`) и добавлять в список `Res`, пока куча не опустеет. При этом пирамида будет перестраиваться, и наибольшее из оставшихся чисел снова будет оказываться на вершине.

```
from heapq import* def heap_sort(A):
    heapify(A)
    Res = []
    while len(A) != 0:
        x = heappop(A)
        Res.append(x)
    return Res
```

Пирамида устроена таким образом, что количество ступеней равно $log_2 N$, а добавление и извлечение числа также пропорционально этой величине. В целом асимптотика алгоритма становится равной $O(N \cdot \log_2 N)$.

Однако последней сортировки есть серьезный недостаток – она неустойчива.

Устойчивость сортировки

Сортировка называется устойчивой, если равные друг другу элементы остаются в исходном порядке.

Лекция № 12

Т. Ф. Хирьянов

Объектно-ориентированное программирование

Объектно-ориентированное программирование держится на трех китах. Это

1. Инкапсуляция
2. Полиморфизм
3. Наследование

Модульная парадигма является частью инкапсуляции и заключается в объединении в группы функций с некоторым общим назначением (например, в один модуль удобно собрать функции, задействованные в вводе и интерпретации данных). Такая парадигма широко распространена в не объектно-ориентированных языках программирования.

Следующим шагом к инкапсуляции является введение *классов* Это структуры (типы) данных, которые обладают определенными *атрибутами*, а также *методами* – функциями с помощью которых осуществляется доступ к атрибутам.

Возможно существование нескольких экземпляров класса – объектов. Так могут существовать два объекта: заяц Вася и заяц Дима. Оба они будут являться экземплярами класса заяц (Rabbit). Заяц имеет два свойства (атрибута): размер (size) и сытость (sytost). К ним можно обратиться с помощью методов класса: set_size, устанавливающего определенный размер size, get_size, возвращающего значение size, и feed, увеличивающего сытость. Все эти функции имеют обязательный аргумент self, стоящий на первом месте. Он является ссылкой на экземпляр класса, и должен быть обязательно указан в описании метода (только так метод «поймет», для которого экземпляра класса он вызван).

```
class Rabbit:
    size = None
    sytost = 0
    def set_size(self, size):
        self.size = size
    def get_size(self):
        return self.size
    def feed(self, meal):
        self.sytost += calorii(meal)
```

Стоит отметить, что обращаться к атрибутам и манипулировать ими удобнее не на прямую, а с помощью методов класса. Тогда можно производить действия с атрибутами, используя понятный интерфейс, не задумываясь, как именно они осуществляются. Из приведенного ниже примера ясно, что кролика a покормили морковкой. При этом мы не знаем, что это подразумевает под собой.

```
a = Rabbit()
b = Rabbit()
a.feed(carrot)       # Rabbit.feed(a, carrot)
```

Полиморфизм в Python

В питоне невозможно перегружать функции. В рассмотренном ниже примере происходит не перегрузка, а переопределение функции f – старое определение пропадает.

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

Для того чтобы обойти это препятствие можно воспользоваться значениями по умолчанию. Допустим, что функция f должна возвращать произведение двух поданных на вход чисел и квадрат числа, если подано лишь одно число. Для обработки последней ситуации удобно определить по умолчанию один из аргументов значением None.

```
def f(x, y=None):
    if y == None:
        y = x
    return x*y
```

Можно однако использовать аргументы разного типа. И внутри функции определить, как вести себя с теми или иными объектами. Этим и достигается полиморфизм в Python.

setter и getter

Создадим класс студента. У него необходимо создать атрибуты возраст и имя, причем они должны определяться автоматически при создании объекта класса student. Это делается с помощью специального метода-конструктора __init__. Для того чтобы сделать доступ к атрибуту извне нежелательным, его имя должно начинаться с символа «_» (тогда данного атрибута не будет видно в списке атрибутов класса). Доступ к атрибуту обычно осуществляется с помощью функции «getter», которая имеет такое же название, что и атрибута и перед ее определением написано `@property`. Для того чтобы установить желаемое значение аргумента, правильнее использовать setter. Его определение аналогично.

```
class Student:
    def __init__(self, age, name):   # Конструктор
        self._age = age
        self._name = name
    def aging(self):
        self._age += 1
        print('Ура! Мне', self._age, 'лет!')
    @property
    def age(self):                   # «getter» (свойство)
    return self._age
    @age.setter
    def age(self, age):
        self.age = age
```

Тогда можно возвратить значение атрибута показанным ниже способом (здесь осуществляется не прямой доступ к атрибуту, а с помощью функции getter)

```
a = Student(17, 'Вася')
a.age = 18   # Нельзя!
x = a.age
```

Определение функций

Важно отметить, что к объектам класса можно приписывать новые атрибуты уже после определения класса. Так как любая функция является объектом класса функций, то и для нее можно создать атрибуты. Например, атрибут p функции productor можно задать уже после объявления функции. Причем она будет работать только после этого.

```
def productor(x):
    return x*productor.p
productor.p = 5
print(productor(2))     # 10
```

Рассмотрим класс двумерного вектора Vector2D. Допустим мы хотим сложить два вектора не с помощью некоторой функции, а привычным способом (c = a + b). Для этого нужно определить специальную функцию __add__. Таким же образом можно организовать умножение (__mul__). Необходимо однако различать скалярное произведение и умножение вектора на число. Это делается с помощью проверки принадлежности (isInstance) объекта тому или иному типу (или его прямому потомку).

```
class Vector2D:
    def __init__(self, x=0, y=0):
        self._x = x
        self._y = y
    def __add__(self, other):
        x = self._x + other._x
        y = self._y + other._y
        return Vector2D(x, y)
    def __mul__(self, other):
        if isInstance(other, Vector2D):
            return self._x *other._x + self._y *other._y
        else:
            return Vector2D(self._x * other, self._y * other)
```

range

Допустим, что нужно создать итератор, который бы не сразу создавал весь список, а возвращал бы один элемент и останавливался на паузу. Например, нужно генерировать геометрическую прогрессию. Это можно осуществить следующим образом. Приведенная ниже функция является функцией-генератором. Вместо return используется yield.

```
def geom_range(start, stop, d):
    x = start
    while x < stop:
        yield x
        x *= d
```

Затем данную функцию можно использовать как итератор.

```
for x in geom_range(1, 1025, 2):
    print(x)
```

Многие функции умеют работать с данными итераторами.

```
g_num = dict(zip(range(0, 11), geom_range(1, 1025, 2))
x, y, z = list(map(int, input().split())
x, y, z = [int(x) for x in input().split()]
```

Списки level up

Очевидно, что в списках могут храниться другие списки. Однако в действительности там хранятся ссылки на объекты. В приведенном ниже примере последний элемент списка и вовсе хранит ссылку на сам список.

```
a = [1]
a.append(a)  # [1, *]
x = a
for i in range(10):
    print(x[0])
    x = x[1]
```

Данный алгоритм может быть основой для структуры данных, представляющей собой систему вложенных списков, таких что последний элемент n - 1 списка содержит ссылку на n список. Это *односвязный* список. Его преимущество состоит в том, что сложность включения нового элемента, например, в середину списка составляет O(1), в отличие от O(N) для массива.

Лекция № 13

Т. Ф. Хирьянов

Очередь (queue)

Очередь – это структура данных, в которую можно добавлять элемент в ее конец и извлекать элемент из ее начала. Одним из простых видов очереди является FIFO (First Input – First Out). В такой очереди первый вошедший элемент является первым претендентом на выход.

Очередь требуется, когда наблюдается недостаток ресурсов. Например, если некоторая программа не может обрабатывать больше двадцати запросов в секунду, то на двадцать первый запрос очередь переполнится, т.е. запросы перестанут попадать в нее.

В стандартной библиотеке Python есть модуль queue, который позволяет создавать очередь и работать с нею.

Создается очередь, как и любой объект класса, с помощью вызова конструктора Queue. Ему можно передать количество элементов, которые можно хранить в памяти одновременно. Можно узнать пуста ли очередь с помощью метода empty и проверить на полноту, используя метод full (они возвращают правду или ложь). Чтобы положить элемент в очередь нужно воспользоваться методом put, а для того чтобы извлечь – get.

```
import queue
q = Queue(10)  # очередь из 10 элементов
q.empty()
q.full
q.put(item)
q.get()
```

Попробуем реализовать класс Queue, используя двусвязный список, с максимальным количеством звеньев, равным количеству элементов в очереди. Реализуем конструктор. По умолчанию очередь создается пустой, и начало (_begin) и конец (_end) указывают на None; максимальное количество элементов (N_max) можно задать в конструкторе, однако по умолчанию оно равно нулю; еще один атрибут (N) – это текущее количество элементов в очереди.

Методы empty и full реализуются очевидным образом. Во время выполнения операции добавления нового элемента в случае, когда очередь уже полна, будет вызвана ошибка. Более того, если очередь пуста, то операция будет работать иначе, чем когда в очереди уже есть элементы. Так как список двусвязный, то необходимо связать не только новый первый элемент со старым, но и наоборот.

Если очередь пуста, то метод извлечения элемента get вызовет ошибку. Если она не пуста, то последний элемент можно извлечь. При этом необходимо сначало сохранить ссылку на этот элемент, прежде чем у предпоследнего элемента в качестве ссылки на последний указать None. Важно обратить внимание, что сборщик мусора удалит элемент, только когда на него не останется ни одной ссылки. В случае когда в очереди был всего один элемент, необходимо также удалить ссылку на него как на начало очереди.

```
class Queue:
    def __init__(self, N=0):
        self._begin = None
        self._end = None
        self._N_max = N
        self._N = N

    def empty(self):
        return self._N == 0

    def full (self):
        return self._N == self._N_max if self._N_max != 0 else False

    def put(self, item):
        if self.full():
            raise Error()         # вызов ошибки при переполнении
        if self.empty():
            self._begin = self._end = [None, item, None]
            # ставим None, так как предыдущего и следующего звена нет
        else:
            new_node = (None, item, self._begin)
            self._begin = new_node
            # теперь новый первый элемент связан со старым первым элементом
            self._begin[2][0] = self._begin
            # теперь старый первый элемент связан с новым первым элементом
        self._N += 1              # количество элементов увеличилось на единицу

    def get(self):
        if self.empty():
            raise Error()
        item = self._end[1]
            # теперь последний элемент сохранен
        self._end = self._end[0]
            # теперь последним является ранее предпоследний элемент
        if self._end == None:     # т.е. был всего один элемент
            self._begin = None
        else:
            self._end[2] = None   # так как он теперь последний
            # теперь ссылка на последний элемент будет утеряна во всех случаях
        self._N -= 1
```

Куча (heap)

Куча или пирамида представляет собой двоичное дерево, в котором самый большой элемент находится на самом верху. Каждый элемент пирамиды пронумерован в порядке сверху вниз и слева направо. На самой верхней ступени один элемент, на второй – два, на третьей – четыре и т.д. Нулевой элемент является родительским для первого и второго, первый – для третьего и четвертого, второй – для пятого и шестого. В общем случае для элемента с индексом i родительским является элемент индексом j = (i - 1) // 2. Тем самым пирамиду можно хранить в виде обычного списка: место элемента в пирамиде будет определяться его индексом. В пирамиде поддерживается

следующая упорядоченность: каждый дочерний элемент не больше родительского. Если количество элементов равно нулю или одному, то, очевидно, ничего делать не требуется, однако если их больше, то при добавлении нового элемента (обычно в конец списка) нужно определить место последнего в родительской цепочке, сравнивая их и, если потребуется, меняя местами.

В качестве примера реализуем класс Heap только лишь с конструктором и методом вставки нового элемента put.

```
class Heap:
    def __init__(self):
        self._m = []
    def put(self, item):
        self._m.append(item)
        i = len(self._m) - 1
        j = (i - 1) // 2
        while i != 0 and self._m[i] > self._m[j]:
            self._m[i], self._m[j] = self._m[j], self._m[i]
            i = j
            j = (i - 1) // 2
```

Очередь LIFO или стек(stack)

Стек представляет собой очередь, в которой последний вошедший является первым претендентом на выход (Last In – First Out). Для стека характерны метод пополнения стека push и метод извлечения последним вошедшего элемента pop, чья реализация является очевидной.

```
class Stack:
    def __init__(self):
        self._m = []
    def push(self, item):
        self._m.append(item)
    def pop(self):
        top = self._m[-1]
        self._m.pop()
        return top
```

Стек применяется для правильного взаимоотношения вызываемых функций (стек вызовов).

Исключения

В процессе работы программы часто возникают исключительные случаи. Например, это может быть деление на ноль или некорректный ввод (вместо цифры случайно нажимается клавиша с буквой). Для обработки исключений существует блок try ... except.

```
def f():
    try:
        # команды нормального выполнения
    except:
        # команды исключения
```

После except можно указать, какую конкретную ошибку следует обрабатывать (например, TypeError или ZeroDivisionError). Можно также указать несколько обработчиков для разных типов ошибок (это напоминает блок if ... elif ... elif ... else).

Если несколько функций вызывают одна другую, то каждая из них может обрабатывать какие-нибудь из возможных ошибок. При возникновении ошибки, она будет «искать» свой обработчик, двигаясь по стеку вызовов от самой последней вызванной функции до первой, пока не натолкнется на него. Если же этого не произойдет, то будет осуществлен выход из всей программы с сообщением о конкретной ошибке.

Лекция № 13

Т. Ф. Хирьянов

Хэширование. Хэш-функция

Представим себе экзамен. У каждого студента есть зачетная книжка. Семинарист подписал каждому конкретному студенту конкретный билет. Студенты положили свои зачетки на стол в произвольном порядке. Получилось два списка: фамилий и номеров билетов. Ставится задача вложить в каждую зачетку соответствующий билет и оценить сложность решения для различных алгоритмов (количество студентов равно N).

Первый способ состоит в том, что для каждой следующей зачетки просматриваются все оставшиеся фамилии. Это полный перебор, и его сложность $O(N^2)$.

Второй способ состоит в том, что можно отсортировать оба списка по ключам. Тогда асимптотика соответствует скорости сортировки (например, может быть достигнута скорость $O(N \cdot \ln_2 N)$).

Однако наиболее эффективным является третий способ. Записать все фамилии и соответствующие им билеты в словарь (dict), где ключем является фамилия, а значением – номер билета. Сложность создания словаря составит $O(N)$, а сложность поиска в нем – $O(1)$ (т.е. за конечное количество операций не зависящих от количества элементов).

Рассмотрим другую задачу. Каким образом лучше всего организовать телефонную книгу, чтобы быстрее всего по номеру определить абонента?

Один из способов – отсортировать кортежи (номер, фамилия) по номерам. Тогда задача решится бинарным поиском и сложность составит $O(\ln_2 N)$. Это хороший способ, однако не идеальный.

Пусть например есть записная книжка из 100 страниц. Тогда каждого нового абонента можно записывать на страницу с номером, равным остатку от деления номера абонента на 100 (т.е. запись «12345 – Вася» попадет на 45 страницу). Такой подход называется *хэшированием*: ключу (номеру) сопоставляется хэш-функция (остаток от деления).

Коллизии

Так как у пятизначных номеров (в разобраном выше примере) мощность ключей - 10^5, а мощность множества индексов (количество страниц) 10^2, то, очевидно, возможна *коллизия* – на одну страницу попадут несколько номеров. Если одной хэш-функции соответствует очень много ключей то имеет место массовая коллизия. Поэтому для для хэш-функций существует следующее требование: для близко лежащих значений они должны давать существенно различные результаты. Еще одним требованием является скорость работы хэш-функции. Если она низкая, то хэширование теряет свое преимущество.

Есть два пути разрешения коллизий: открытая и закрытая хэш-таблицы. В первом случае разрешается записывать несколько номеров на странице – получается словарь списков. Во втором случае это запрещается и «лишние» номера записываются в следующую ячейку.

Если нарисовать асимптотику скорости поиска в открытой хэш-таблице от количества хранимых данных, то вначале она будет константой, а по мере приближения к максимальному количеству ячеек будут возникать коллизии и время поиска будет увеличиваться; когда это значение будет пройдено асимптотика станет линейной.

Для закрытой хэш-таблицы зависимость практически аналогична, однако по достижении максимума больше в нее будет нельзя ничего добавить.

Открытая хэш-таблица

Описанную выше открытую хэш-таблицу можно реализовать следующим образом.

```
Class MyDict:
    N_max = 100  # максимальный размер таблицы

    def add(self, key, value):
        i = hash(key) % N_max
        j = 0
        while j < len(self._M[i]):                # пока i-й список не закончился
            if self._M[i][j][0] == key:           # ключ j-ого кортежа в i-ом списке.
                self._M[i][j] = (key, value)      # перезаписываем значение для ключа
                break
        else:                                     # вышли из цикла, но break не было
            self._M[i].append((key, value))       # можно записать пару ключ-значение

    def __init__(self, lst=[]):
        self._M = [ [] for I in range(N_max)]     # список пустых списков
        for i, value in lst:        # бежит по парам ключ-значение поданного списка
            self.add(key, value)



    def get(self, key):
        i = hash(key) % N_max
        for k, v in self._M[i]:     # поиск в i-ом списке пар
            if k == key:
                return v
            return None             # хотя лучше вызвать исключение



    def del(self, key):
        i = hash(key) % N_max
        j = 0
        while j < len(self._M[i]):           # пока i-й список не закончился
            if self._M[i][j][0] == key:      # ключ j-ого кортежа в i-ом списке.
                self._M[i].pop(j)            # удаляем пару ключ-значение
                return
        raise KeyError()                     # если удалять нечего
```