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.

$A238BE73 \longrightarrow 100$

$B202CC84 \longrightarrow 200$

$B555PH71 \longrightarrow 1000$

...

$A999AA100$

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)]
```

Или такого рода таблицу.

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

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')

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

```
    x = input()
while A:
    print(A.pop())
```