Лекционное занятие

по учебной дисциплине

Специализированные языки программирования

на тему:

Основы языка программирования Python 3

План занятия

1. Основные понятия языка Python
2. Основные логические конструкции
3. Функции
4. Списки
5. Словари
6. Строки

Основные понятия языка Python

python™

Python интерпретируемый объектно-ориентированный язык программирования высокого уровня со строгой динамической типизацией. Разработан в 1990 году Гвидо ван Россум. Python поддерживает модули и пакеты модулей, способствует модульности и повторному использованию кода. Интерпретатор Python и стандартные библиотеки доступны как в откомпилированной так и в исходной форме на всех основных платформах. В языке программирования Python поддерживаются несколько парадигм программирования, в частности: объектно-ориентированная, процедурная, функциональная и аспектно-ориентированная.

```
print("Привет, Мир!") # print -- это функция
#  print -- это функция
print('Привет, Мир!')
```

Комментарии – это то, что пишется после символа #, и представляет интерес лишь как заметка для читающего программу.

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2 + 2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

Математические операторы

Оператор	Описание	Пример	Результат
+	Сложение	7 + 3	10
-	Вычитание	7 - 3	4
*	Умножение	7 * 3	21
/	Деление (истинное)	7 / 3	2.3333333333333335
**	Возведение в степень	7**3	343
//	Целочисленное деление	7 // 3	2
%	Остаток от деления	7 % 3	1

Тип обьекта	Пример
Числа	123, 14.32, 3+123
Строки	spam', "was'nt"
Списки	[1,['two','three'],4,[5]]
Словари	{'room':'bed', 'chiken':'bird'}
Кортежи	(2,'big world',4,'five')
Файлы	filename=open('file.txt','r')
Множества	set('abc'), {'a','b','c'}

Числа в Python бывают следующих типов: целые, с плавающей точкой и комплексные.

1.Примером целого числа может служить 2.
2.Примерами чисел с плавающей точкой (или “плавающих” для краткости) могут быть 3.23 и 52.3E-4. Обозначение E показывает степени числа 10. В данном случае 52.3E-4 означает 52.3 * 10-4.
3.Примеры комплексных чисел: (-5+4j) и (2.3 - 4.6j)

Числа записываются последовательностью цифр, также перед числом может стоять знак минус, а строки записываются в одинарных кавычках. 2 и '2' — это разные объекты, первый объект — число, а второй —строка.
Операция + для целых чисел и для строк работает по-разному: для чисел это сложение, а для строк —конкатенация.

Другой класс чисел - действительные (вещественные числа), представляемые в виде десятичных дробей. Они записываются с использованием десятичной точки, например, 2.0. В каком-то смысле, 2 и 2.0 имеют равные значение, но это—разные объекты. Например, можно вычислить значения выражения 'ABC' * 10 (повторить строку 10 раз), но нельзя вычислить 'ABC' * 10.0.

Определить тип объекта можно при помощи функции type:

```
>>> type(2)
<class 'int'>
>>> type('2')
<class 'str'>
>>> type(2.0)
<class 'float'>
```

Обратите внимание —type является функцией, аргументы функции указываются в скобках после ее имени.

Вот список основных операций для чисел:
A + B — сумма;
A - B — разность;
A * B — произведение;
A / B — частное;
A ** B — возведение в степень. Полезно помнить, что квадратный корень из числа x — это x ** 0.5, а корень степени n это x ** (1 / n).

Есть также унарный вариант операции -, то есть операция с одним аргументом. Она возвращает число, противоположное данному. Например: -A.

В выражении может встречаться много операций подряд. Как в этом случае определяется порядок действий? Например, чему будет равно 1 + 2 * 3 ** 1 + 1? В данном случае ответ будет 8, так как сначала выполняется возведение в степень, затем – умножение, затем — сложение.

Более общие правила определения приоритетов операций такие:

1.Выполняются возведения в степень справа налево, то есть 3 ** 3 ** 3 это 3^27

2.Выполняются унарные минусы (отрицания).

3.Выполняются умножения и деления слева направо. Операции умножения и деления имеют одинаковый приоритет.

4.Выполняются сложения и вычитания слева направо. Операции сложения и вычитания имеют одинаковый приоритет.

Основные логические конструкции

Синтаксис условной инструкции

Все ранее рассматриваемые программы имели линейную структуру: все инструкции выполнялись последовательно одна за одной, каждая записанная инструкция обязательно выполняется.

Допустим мы хотим по данному числу x определить его абсолютную величину (модуль). Программа должна напечатать значение переменной x, если x>0 или же величину -x в противном случае. Линейная структура программы нарушается: в зависимости от справедливости условия x>0 должна быть выведена одна или другая величина. Соответствующий фрагмент программы на Питоне имеет вид:

```
x = int(input())
if x > 0:
    print(x)
else:
    print(-x)
```

В этой программе используется условная инструкция if (если). После слова if указывается проверяемое условие (x > 0), завершающееся двоеточием. После этого идет блок (последовательность) инструкций, который будет выполнен, если условие истинно, в нашем примере это вывод на экран величины x. Затем идет слово else (иначе), также завершающееся двоеточием, и блок инструкций, который будет выполнен, если проверяемое условие неверно, в данном случае будет выведено значение -x.

Итак, условная инструкция в Пайтоне имеет следующий синтаксис:

```
if Условие:
    Блок инструкций 1
else:
    Блок инструкций 2
```

Блок инструкций 1 будет выполнен, если Условие истинно. Если Условие ложно, будет выполнен Блок инструкций 2.

В условной инструкции может отсутствовать слово else и последующий блок. Такая инструкция называется неполным ветвлением. Например, если дано число x и мы хотим заменить его на абсолютную величину x, то это можно сделать следующим образом:

```
x = int(input("Enter the number: "))
y = "null"
if x < 0:
    y = -x
print("y= " + str(y))
```

В этом примере переменной x будет присвоено значение -x, но только в том случае, когда x<0. А вот инструкция print(x) будет выполнена всегда, независимо от проверяемого условия.

Для выделения блока инструкций, относящихся к инструкции if или else в языке Питон используются отступы. Все инструкции, которые относятся к одному блоку, должны иметь равную величину отступа, то есть одинаковое число пробелов в начале строки. Рекомендуется использовать отступ в 4 пробела и не рекомедуется использовать в качестве отступа символ табуляции.

Вложенные условные инструкции

Внутри условных инструкций можно использовать любые инструкции языка Пайтон, в том числе и условную инструкцию. Получаем вложенное ветвление – после одной развилки в ходе исполнения программы появляется другая развилка. При этом вложенные блоки имеют больший размер отступа (например, 8 пробелов). Покажем это на примере программы, которая по данным ненулевым числам x и y определяет, в какой из четвертей координатной плоскости находится точка (x,y):

```
x = int(input("Введите x: "))
y = int(input("Введите y: "))
if x >= 0:
    if y >= 0:              # x>0, y>0
        print("Первая четверть")
    else:                   # x>0, y<0
        print("Четвертая четверть")
else:
    if y >= 0:              # x<0, y>0
        print("Вторая четверть")
    else:                   # x<0, y<0
        print("Третья четверть")
```

В этом примере мы использовали комментарии – текст, который интерпретатор игнорирует. Комментариями в Питоне является символ # и весь текст после этого символа до конца строки.

Операторы сравнения

Как правило, в качестве проверяемого условия используется результат вычисления одного из следующих операторов сравнения:

<

Меньше — условие верно, если первый операнд меньше второго.

>

Больше — условие верно, если первый операнд больше второго.

<=

Меньше или равно.

>=

Больше или равно.

==

Равенство. Условие верно, если два операнда равны.

!=

Неравенство. Условие верно, если два операнда неравны.

Например, условие (x * x < 1000) означает “значение x * x меньше 1000”, а условие (2 * x != y) означает “удвоенное значение переменной x не равно значению переменной y”.

Операторы сравнения в Пайтоне можно объединять в цепочки (в отличии от большинства других языков программирования, где для этого нужно использовать логические связки), например, a == b == c или 1 <= x <= 10.

Тип данных bool

Операторы сравнения возвращают значения специального логического типа bool. Значения логического типа могут принимать одно из двух значений: True (истина) или False (ложь). Если преобразовать логическое True к типу int, то получится 1, а преобразование False даст 0. При обратном преобразовании число 0 преобразуется в False, а любое ненулевое число в True. При преобразовании str в bool пустая строка преобразовывается в False, а любая непустая строка в True.

Логические операторы

Иногда нужно проверить одновременно не одно, а несколько условий. Например, проверить, является ли данное число четным можно при помощи условия (n % 2 == 0) (остаток от деления n на 2 равен 0), а если необходимо проверить, что два данных целых числа n и m являются четными, необходимо проверить справедливость обоих условий: n % 2 == 0 и m % 2 == 0, для чего их необходимо объединить при помощи оператора and (логическое И): n % 2 == 0 and m % 2 == 0.

В Пайтоне существуют стандартные логические операторы: логическое И, логическое ИЛИ, логическое отрицание.

Логическое И является бинарным оператором (то есть оператором с двумя операндами: левым и правым) и имеет вид and. Оператор and возвращает True тогда и только тогда, когда оба его операнда имеют значение True.

Логическое ИЛИ является бинарным оператором и возвращает True тогда и только тогда, когда хотя бы один операнд равен True. Оператор “логическое ИЛИ” имеет вид or.

Логическое НЕ (отрицание) является унарным (то есть с одним операндом) оператором и имеет вид not, за которым следует единственный операнд. Логическое НЕ возвращает True, если операнд равен False и наоборот.

Пример. Проверим, что хотя бы одно из чисел a или b оканчивается на 0:

```
if a % 10 == 0 or b % 10 == 0:
```

Проверим, что число a — положительное, а b — неотрицательное:

```
if a > 0 and not (b < 0):
```

Или можно вместо not (b < 0) записать (b >= 0).

Каскадные условные инструкции

Пример программы, определяющий четверть координатной плоскости, можно переписать используя “каскадную“ последовательность операцией if... elif... else:

```
x = int(input("Введите x: "))
y = int(input("Введите y: "))
if x >= 0 and y >= 0:
    print("Первая четверть")
elif x >= 0 and y < 0:
    print("Четвертая четверть")
elif y >= 0:
    print("Вторая четверть")
else:
    print("Третья четверть")
```

В такой конструкции условия if, ..., elif проверяются по очереди, выполняется блок, соответствующий первому из истинных условий. Если все проверяемые условия ложны, то выполняется блок else, если он присутствует.

Цикл for

Цикл for, также называемый циклом с параметром, в языке Пайтон богат возможностями. В цикле for указывается переменная и множество значений, по которому будет пробегать переменная. Множество значений может быть задано списком, кортежем, строкой или диапазоном.

Вот простейший пример использования цикла, где в качестве множества значений используется кортеж:

```
i = 1
for color in ('red', 'orange', 'yellow', 'green', 'cyan', 'blue', 'violet'):
    print(str(i) + '-th color of rainbow is '+ color, sep = '')
    i += 1
```

В этом примере переменная color последовательно принимает значения 'red', 'orange' и т.д. В теле цикла выводится сообщение, которое содержит название цвета, то есть значение переменной color, а также номер итерации цикла число, которое сначала равно 1, а потом увеличивается на один (инструкцией i += 1 с каждым проходом цикла.

В списке значений могут быть выражения различных типов, например:

```
for i in 1, 2, 3, 'one', 'two', 'three':
    print(i)
```

При первых трех итерациях цикла переменная i будет принимать значение типа int, при последующих трех — типа str.

Цикл while

Цикл while (“пока”) позволяет выполнить одну и ту же последовательность действий, пока проверяемое условие истинно. Условие записывается до тела цикла и проверяется до выполнения тела цикла. Как правило, цикл while используется, когда невозможно определить точное значение количества проходов исполнения цикла.

Синтаксис цикла while в простейшем случае выглядит так:

```
while условие:
    блок инструкций
```

При выполнении цикла while сначала проверяется условие. Если оно ложно, то выполнение цикла прекращается и управление передается на следующую инструкцию после тела цикла while. Если условие истинно, то выполняется инструкция, после чего условие проверяется снова и снова выполняется инструкция. Так продолжается до тех пор, пока условие будет истинно. Как только условие станет ложно, работа цикла завершится и управление передастся следующей инструкции после цикла.

Например, следующий фрагмент программы напечатает на экран всех целые числа от 1 до 10. Видно, что цикл while может заменять цикл for ... in range(...):

```
i = 1
while i <= 10:
    print(i)
    i += 1
```

В этом примере переменная i внутри цикла изменяется от 1 до 10. Такая переменная, значение которой меняется с каждым новым проходом цикла, называется счетчиком. Заметим, что после выполнения этого фрагмента значение переменной i будет равно 11, поскольку именно при i==11 условие i<=10 впервые перестанет выполняться.

Вот еще один пример использования цикла while для определения количества цифр натурального числа n:

```
n = int(input())
length = 0
while n > 0:
    n //= 10
    length += 1
```

В этом цикле мы отбрасываем по одной цифре числа, начиная с конца, что эквивалентно целочисленному делению на 10 (n //= 10), при этом считаем в переменной length, сколько раз это было сделано.

В языке Пайтон есть и другой способ решения этой задачи: length = len(str(i)).

Инструкции управления циклом

После тела цикла можно написать слово else: и после него блок операций, который будет выполнен один раз после окончания цикла, когда проверяемое условие станет неверно:

```
i = 1
while i <= 10:
    print(i)
    i += 1
else:
    print('Цикл окончен, i = ' + str(i))
```

Казалось бы, никакого смысла в этом нет, ведь эту же инструкцию можно просто написать после окончания цикла. Смысл появляется только вместе с инструкцией break, использование которой внутри цикла приводит к немедленному прекращению цикла, и при этом не исполняется ветка else. Разумеется, инструкцию break осмыленно вызывать только из инструкции if, то есть она должна выполняться только при выполнении какого-то особенного условия.

Другая инструкция управления циклом — continue (продолжение цикла). Если эта инструкция встречается где-то посередине цикла, то пропускаются все оставшиеся инструкции до конца цикла, и исполнение цикла продолжается со следующей итерации.

Инструкции break, continue и ветка else: можно использовать и внутри цикла for. Тем не менее, увлечение инструкциями break и continue не поощряется, если можно обойтись без их использования. Вот типичный пример плохого использования инструкции break.

```
while True:
    length += 1
    n //= 10
    if n == 0:
        break
```

Функции

Функции – это многократно используемые фрагменты программы. Они позволяют дать имя определённому блоку команд с тем, чтобы впоследствии запускать этот блок по указанному имени в любом месте программы и сколь угодно много раз. Это называется вызовом функции.

Функции определяются при помощи зарезервированного слова def. После этого слова указывается имя функции, за которым следует пара скобок, в которых можно указать имена некоторых переменных, и заключительное двоеточие в конце строки. Далее следует блок команд, составляющих функцию. На примере можно видеть, что на самом деле это очень просто:

Пример:

```
def sayHello():
    print('Привет, Мир!') # блок, принадлежащий функции
# Конец функции

sayHello() # вызов функции
sayHello() # ещё один вызов функции
```

Мы определили функцию с именем sayHello, используя описанный выше синтаксис. Эта функция не принимает параметров, поэтому в скобках не объявлены какие-либо переменные. Параметры функции – это некие входные данные, которые мы можем передать функции, чтобы получить соответствующий им результат.

Обратите внимание, что мы можем вызывать одну и ту же функцию много раз, а значит нет необходимости писать один и тот же код снова и снова.

Параметры функций

Функции могут принимать параметры, т.е. некоторые значения, передаваемые функции для того, чтобы она что-либо сделала с ними. Эти параметры похожи на переменные, за исключением того, что значение этих переменных указывается при вызове функции, и во время работы функции им уже присвоены их значения.

Параметры указываются в скобках при объявлении функции и разделяются запятыми. Аналогично мы передаём значения, когда вызываем функцию. Обратите внимание на терминологию: имена, указанные в объявлении функции, называются параметрами, тогда как значения, которые вы передаёте в функцию при её вызове, – аргументами.

```
def printMax(a, b):
    if a > b:
        print(a, 'максимально')
    elif a == b:
        print(a, 'равно', b)
    else:
        print(b, 'максимально')

printMax(3, 4) # передача переменных в качестве аргументов
```

Локальные переменные

При объявлении переменных внутри определения функции, они никоим образом не связаны с другими переменными с таким же именем за пределами функции – т.е. имена переменных являются локальными в функции. Это называется областью видимости переменной. Область видимости всех переменных ограничена блоком, в котором они объявлены, начиная с точки объявления имени.

Пример:

```
x = 50
def func(x):
    print('x равен', x)
    x = 2
    print('Замена локального x на ' + str(x))

func(x)
print('x по-прежнему' + str(x))
```

Вывод:

```
$ python func_local.py
x равен 50
Замена локального x на 2
x по-прежнему 50
```

Как это работает:

При первом выводе значения, присвоенного имени x, в первой строке функции Python использует значение параметра, объявленного в основном блоке, выше определения функции.

Далее мы назначаем x значение 2. Имя x локально для нашей функции. Поэтому когда мы заменяем значение x в функции, x, объявленный в основном блоке, остаётся незатронутым.

Последним вызовом функции print мы выводим значение x, указанное в основном блоке, подтверждая таким образом, что оно не изменилось при локальном присваивании значения в ранее вызванной функции.

Зарезервированное слово “global”

Чтобы присвоить некоторое значение переменной, определённой на высшем уровне программы (т.е. не в какой-либо области видимости, как то функции или классы), необходимо указать Python, что её имя не локально, а глобально (global). Сделаем это при помощи зарезервированного слова global. Без применения зарезервированного слова global невозможно присвоить значение переменной, определённой за пределами функции.

Можно использовать уже существующие значения переменных, определённых за пределами функции (при условии, что внутри функции не было объявлено переменной с таким же именем). Однако, это не приветствуется, и его следует избегать, поскольку человеку, читающему текст программы, будет непонятно, где находится объявление переменной. Использование зарезервированного слова global достаточно ясно показывает, что переменная объявлена в самом внешнем блоке.

Пример:

```
x = 50
def func():
    global x

    print('x равно', x)
    x = 2
    print('Заменяем глобальное значение x на', x)

func()
print('Значение x составляет', x)
```

Вывод:

```
$ python func_global.py
x равно 50
Заменяем глобальное значение x на 2
Знчение x составляет 2
```

Как это работает:

Зарезервированное слово global используется для того, чтобы объявить, что x – это глобальная переменная, а значит, когда мы присваиваем значение имени x внутри функции, это изменение отразится на значении переменной x в основном блоке программы.

Значения аргументов по умолчанию

Зачастую часть параметров функций могут быть необязательными, и для них будут использоваться некоторые заданные значения по умолчанию, если пользователь не укажет собственных. Этого можно достичь с помощью значений аргументов по умолчанию. Их можно указать, добавив к имени параметра в определении функции оператор присваивания (=) с последующим значением.

Обратите внимание, что значение по умолчанию должно быть константой.

Пример:

```
def say(message, times = 1):
    print(message * times)

say('Привет')
say('Мир', 5)
```

```
Привет
МирМирМирМирМир
```

Как это работает:

Функция под именем say используется для вывода на экран строки указанное число раз. Если мы не указываем значения, по умолчанию строка выводится один раз. Мы достигаем этого указанием значения аргумента по умолчанию, равного 1 для параметра times.

При первом вызове say мы указываем только строку, и функция выводит её один раз. При втором вызове say мы указываем также и аргумент 5, обозначая таким образом, что мы хотим сказать фразу 5 раз.

Важно Значениями по умолчанию могут быть снабжены только параметры, находящиеся в конце списка параметров. Таким образом, в списке параметров функции параметр со значением по умолчанию не может предшествовать параметру без значения по умолчанию. Это связано с тем, что значения присваиваются параметрам в соответствии с их положением. Например, def func(a, b=5) допустимо, а def func(a=5, b) – не допустимо.

Ключевые аргументы

Если имеется некоторая функция с большим числом параметров, и при её вызове требуется указать только некоторые из них, значения этих параметров могут задаваться по их имени – это называется ключевые параметры. В этом случае для передачи аргументов функции используется имя (ключ) вместо позиции (как было до сих пор).

Есть два преимущества такого подхода: во-первых, использование функции становится легче, поскольку нет необходимости отслеживать порядок аргументов; во-вторых, можно задавать значения только некоторым избранным аргументам, при условии, что остальные параметры имеют значения аргумента по умолчанию.

Пример:

```
def func(a, b=5, c=10):
    print('a равно ' + str(a))
    print('b равно ' + str(b))
    print('c равно ' + str(c))

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Вывод:

```
a равно 3, b равно 7, a c равно 10
a равно 25, b равно 5, a c равно 24
a равно 100, b равно 5, a c равно 50
```

Как это работает:

Функция с именем func имеет один параметр без значения по умолчанию, за которым следуют два параметра со значениями по умолчанию.

При первом вызове, func(3, 7), параметр a получает значение 3, параметр b получает значение 7, а c получает своё значение по умолчанию, равное 10.

При втором вызове func(25, c=24) переменная a получает значение 25 в силу позиции аргумента. После этого параметр c получает значение 24 по имени, т.е. как ключевой параметр. Переменная b получает значение по умолчанию, равное 5.

При третьем обращении func(c=50, a=100) мы используем ключевые аргументы для всех указанных значений. Обратите внимание на то, что мы указываем значение для параметра c перед значением для a, даже несмотря на то, что в определении функции параметр a указан раньше c.

Переменное число параметров

Иногда бывает нужно определить функцию, способную принимать любое число параметров. Этого можно достичь при помощи звёздочек:

```
def total(a=5, *numbers, **phonebook):
    print('a= ', a)
    #проход по всем элементам кортежа
    for single_item in numbers:
        print('single_item = ', single_item)
        #проход по всем элементам словаря
    for first_part, second_part in phonebook.items():
        print(first_part,second_part)

print(total(10,1,2,3,Jack=1123,John=2231,Inge=1560))
```

Вывод:

```
a 10
single_item 1
single_item 2
single_item 3
Inge 1560
John 2231
Jack 1123
None
```

Как это работает:

Когда мы объявляем параметр со звёздочкой (например, *param), все позиционные аргументы начиная с этой позиции и до конца будут собраны в кортеж под именем param.

Аналогично, когда мы объявляем параметры с двумя звёздочками (**param), все ключевые аргументы начиная с этой позиции и до конца будут собраны в словарь под именем param.

Только ключевые параметры

Если некоторые ключевые параметры должны быть доступны только по ключу, а не как позиционные аргументы, их можно объявить после параметра со звёздочкой :

```
def total(initial=5, *numbers, extra_number):
    count = initial
    for number in numbers:
        count += number
    count += extra_number
    print(count)

total(10, 1, 2, 3, extra_number=50)
total(10, 1, 2, 3)
# Вызовет ошибку, поскольку мы не указали значение
# аргумента по умолчанию для 'extra_number'.
```

Вывод:

```
66
Traceback (most recent call last):
  File "keyword_only.py", line 12, in <module>
total(10, 1, 2, 3)
TypeError: total() needs keyword-only argument extra_number
```

Как это работает:

Объявление параметров после параметра со звёздочкой даёт только ключевые аргументы. Если для таких аргументов не указано значение по умолчанию, и оно не передано при вызове, обращение к функции вызовет ошибку, в чём мы только что убедились.

Обратите внимание на использование +=, который представляет собой сокращённый оператор, позволяющий вместо x = x + y просто написать x += y.

Если вам нужны аргументы, передаваемые только по ключу, но не нужен параметр со звёздочкой, то можно просто указать одну звёздочку без указания имени: def total(initial=5, *, extra_number).

Оператор “return”

Оператор return используется для возврата из функции, т.е. для прекращения её работы и выхода из неё. При этом можно также вернуть некоторое значение из функции.

Пример:

```
#!/usr/bin/python
# Filename: func_return.py

def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'Числа равны.'
    else:
        return y

print(maximum(2, 3))
```

Вывод:

```
3
```

Как это работает:

Функция maximum возвращает максимальный из двух параметров, которые в данном случае передаются ей при вызове. Она использует обычный условный оператор if..else для определения наибольшего числа, а затем возвращает это число.
Обратите внимание, что оператор return без указания возвращаемого значения эквивалентен выражению return None. None – это специальный тип данных в Python, обозначающий ничего. К примеру, если значение переменной установлено в None, это означает, что ей не присвоено никакого значения.

Каждая функция содержит в неявной форме оператор return None в конце, если вы не указали своего собственного оператора return. В этом можно убедиться, запустив print(someFunction()), где функция someFunction – это какая-нибудь функция, не имеющая оператора return в явном виде. Например:

```
def someFunction():
    pass
```

Оператор pass используется в Python для обозначения пустого блока команд.

Примечание Существует встроенная функция max, в которой уже реализован функционал “поиск максимума”, так что пользуйтесь этой встроенной функцией, где это возможно.

Списки

Списки

Большинство программ работает не с отдельными переменными, а с набором переменных. Например, программа может обрабатывать информацию об учащихся класса, считывая список учащихся с клавиатуры или из файла, при этом изменение количества учащихся в классе не должно требовать модификации исходного кода программы.

Во многих задачах нужно сохранять всю последовательность, например, если бы нам требовалось вывести все элементы последовательности в возрастающем порядке (“отсортировать последовательность”).

Для хранения таких данных можно использовать структуру данных, называемую в Питоне список (в большинстве же языков программирования используется другой термин “массив”). Список представляет собой последовательность элементов, пронумерованных от 0, как символы в строке. Список можно задать перечислением элементов списка в квадратных скобках, например, список можно задать так:

```
Primes = [2, 3, 5, 7, 11, 13]
Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']
```

В списке Primes — 6 элементов, а именно, Primes[0] == 2, Primes[1] == 3, Primes[2] == 5, Primes[3] == 7, Primes[4] == 11, Primes[5] == 13. Список Rainbow состоит из 7 элементов, каждый из которых является строкой.

Также как и символы строки, элементы списка можно индексировать отрицательными числами с конца, например, Primes[-1] == 13, Primes[-6] == 2.

Длину списка, то есть количество элементов в нем, можно узнать при помощи функции len, например, len(A) == 6.

Рассмотрим несколько способов создания и считывания списков. Прежде всего можно создать пустой список (не содержащий элементов, длины 0), в конец списка можно добавлять элементы при помощи метода append. Например, если программа получает на вход количество элементов в списке n, а потом n элементов списка по одному в отдельной строке, то организовать считывание списка можно так:

```
A = []
for i in range(int(input()):
    A.append(int(input())
```

В этом примере создается пустой список, далее считывается количество элементов в списке, затем по одному считываются элементы списка и добавляются в его конец.

Для списков целиком определены следующие операции: конкатенация списков (добавление одного списка в конец другого) и повторение списков (умножение списка на число). Например:

```
A = [1, 2, 3]
B = [4, 5]
C = A + B
D = B * 3
```

В результате список C будет равен [1, 2, 3, 4, 5], а список D будет равен [4, 5, 4, 5, 4, 5]. Это позволяет по-другому организовать процесс считывания списков: сначала считать размер списка и создать список из нужного числа элементов, затем организовать цикл по переменной i начиная с числа 0 и внутри цикла считывается i-й элемент списка:

```
A = [0] * int(input())
for i in range(len(A)):
    A[i] = int(input())
```

Вывести элементы списка A можно одной инструкцией print(A), при этом будут выведены квадратные скобки вокруг элементов списка и запятые между элементами списка. Такой вывод неудобен, чаще требуется просто вывести все элементы списка в одну строку или по одному элементу в строке. Приведем два примера, также отличающиеся организацией цикла:

```
for i in range(len(A)):
    print(A[i])
```

Здесь в цикле меняется индекс элемента i, затем выводится элемент списка с индексом i.

```
for elem in A:
    print(elem, end = ' ')
```

В этом примере элементы списка выводятся в одну строку, разделенные пробелом, при этом в цикле меняется не индекс элемента списка, а само значение переменной (например, в цикле for elem in ['red', 'green', 'blue'] переменная elem будет последовательно принимать значения 'red', 'green', 'blue'.

Методы split и join

Элементы списка могут вводиться по одному в строке, в этом случае строку можно считать функцией input(). После этого можно использовать метод строки split, возвращающий список строк, разрезав исходную строку на части по пробелам. Пример:

```
A = input().split()
```

Если при запуске этой программы ввести строку 1 2 3, то список A будет равен ['1', '2', '3']. Обратите внимание, что список будет состоять из строк, а не из чисел. Если хочется получить список именно из чисел, то можно затем элементы списка по одному преобразовать в числа:

```
for i in range(len(A)):
    A[i] = int(A[i])
```

Используя функции языка map и list то же самое можно сделать в одну строку:

```
A = list(map(int, input().split()))
```

Если нужно считать список действительных чисел, то нужно заменить тип int на тип float.

У метода split есть необязательный параметр, который определяет, какая строка будет использоваться в качестве разделителя между элементами списка. Например, метод split('.') вернет список, полученный разрезанием исходной строки по символам '.'.

Используя “обратные” методы можно вывести список при помощи однострочной команды. Для этого используется метод строки join. У этого метода один параметр: список строк. В результате получается строка, полученная соединением элементов списка (которые переданы в качестве параметра) в одну строку, при этом между элементами списка вставляется разделитель, равный той строке, к которой применяется метод. Например программа

```
A = ['red', 'green', 'blue']
print(' '.join(A))
print(''.join(A))
print('***'.join(A))
```

выведет строки 'red green blue', redgreenblue и red***green***blue.

Если же список состоит из чисел, то придется использовать еще и функцию map. То есть вывести элементы списка чисел, разделяя их пробелами, можно так:

```
print(' '.join(map(str, A)))
```

Генераторы списков

Для создания списка, заполненного одинаковыми элементами, можно использовать оператор повторения списка, например:

```
A = [0] * n
```

Для создания списков, заполненных по более сложным формулам можно использовать генераторы: выражения, позволяющие заполнить список некоторой формулой. Общий вид генератора следующий:

```
[ выражение for переменная in список]
```

где переменная — идентификатор некоторой переменной, список — список значений, который принимает данная переменная (как правило, полученный при помощи функции range), выражение — некоторое выражение, которым будут заполнены элементы списка, как правило, зависящее от использованной в генераторе переменной.

Вот несколько примеров использования генераторов.

Создать список, состоящий из n нулей можно и при помощи генератора:

```
A = [ 0 for i in range(n)]
```

Создать список, заполненный квадратами целых чисел можно так:

```
A = [ i ** 2 for i in range(n)]
```

Если нужно заполнить список квадратами чисел от 1 до n, то можно изменить параметры функции range на range(1, n + 1):

```
A = [ i ** 2 for i in range(1, n + 1)]
```

Вот так можно получить список, заполненный случайными числами от 1 до 9 (используя функцию randint из модуля random):

```
A = [ randint(1, 9) for i in range(n)]
```

А в этом примере список будет состоять из строк, считанных со стандартного ввода: сначала нужно ввести число элементов списка (это значение будет использовано в качестве аргумента функции range), потом — заданное количество строк:

```
A = [ input() for i in range(int(input()))]
```

Словари

Обычные списки (массивы) представляют собой набор пронумерованных элементов, то есть для обращения к какому-либо элементу списка необходимо указать его номер. Номер элемента в списке однозначно идентифицирует сам элемент. Но идентифицировать данные по числовым номерам не всегда оказывается удобно. Например, маршруты поездов в Украине идентифицируются численно-буквенным кодом (число и одна цифра), также численно-буквенным кодом идентифицируются авиарейсы, то есть для хранения информации о рейсах поездов или самолетов в качестве идентификатора удобно было бы использовать не число, а текстовую строку.

Структура данных, позволяющая идентифицировать ее элементы не по числовому индексу, а по произвольному, называется **словарем** или ассоциативным массивом. Соответствующая структура данных в языке Питон называется dict.

Рассмотрим простой пример использования словаря. Заведем словарь Capitals, где индексом является название страны, а значением — название столицы этой страны. Это позволит легко определять по строке с названием страны ее столицу.

```
# Создадим пустой словать Capitals
Capitals = dict()

# Заполним его несколькими значениями
Capitals[‘Poland'] = ‘Warsaw’
Capitals['Ukraine'] = 'Kyiv'
Capitals['USA'] = 'Washington'
```

```
# Считаем название страны
print('В какой стране вы живете?')
country = input()

# Проверим, есть ли такая страна в словаре Capitals
if country in Capitals:
    # Если есть - выведем ее столицу
    print('Столица вашей страны', Capitals[country])
else:
    # Запросим название столицы и добавив его в словарь
    print('Как называется столица вашей страны?')
    city = input()
    Capitals[country] = city
```

Итак, каждый элемент словаря состоит из двух объектов: ключа и значения. В нашем примере ключом является название страны, значением является название столицы. Ключ идентифицирует элемент словаря, значение является данными, которые соответствуют данному ключу. Значения ключей — уникальны, двух одинаковых ключей в словаре быть не может.

В жизни широко распространены словари, например, привычные бумажные словари (толковые, орфографические, лингвистические). В них ключом является слово-заголовок статьи, а значением — сама статья. Для того, чтобы получить доступ к статье, необходимо указать слово-ключ.

Другой пример словаря, как структуры данных — телефонный справочник. В нем ключом является имя, а значением — номер телефона. И словарь, и телефонный справочник хранятся так, что легко найти элемент словаря по известному ключу (например, если записи хранятся в алфавитном порядке ключей, то легко можно найти известный ключ, например, бинарным поиском), но если ключ неизвествен, а известно лишь значение, то поиск элемента с данным значением может потребовать последовательного просмотра всех элементов словаря.

Особенностью ассоциативного массива является его динамичность: в него можно добавлять новые элементы с произвольными ключами и удалять уже существующие элементы. При этом размер используемой памяти пропорционален размеру ассоциативного массива. Доступ к элементам ассоциативного массива выполняется хоть и медленнее, чем к обычным массивам, но в целом довольно быстро.

В языке Пайтон как ключом может быть произвольный неизменяемый тип данных: целые и действительные числа, строки, кортежи. Ключом в словаре не может быть множество, но может быть элемент типа frozenset: специальный тип данных, являющийся аналогом типа set, который нельзя изменять после создания. Значением элемента словаря может быть любой тип данных, в том числе и изменяемый.

Когда нужно использовать словари

Словари нужно использовать в следующих случаях:
Подсчет числа каких-то объектов. В этом случае нужно завести словарь, в котором ключами являются объекты, а значениями — их количество.
Хранение каких-либо данных, связанных с объектом. Ключи — объекты, значения — связанные с ними данные. Например, если нужно по названию месяца определить его порядковый номер, то это можно сделать при помощи словаря Num['January'] = 1; Num['February'] = 2; ....
Установка соответствия между объектами (например, “родитель—потомок”). Ключ — объект, значение — соответствующий ему объект.
Если нужен обычный массив, но при этом масимальное значение индекса элемента очень велико, но при этом будут использоваться не все возможные индексы (так называемый “разреженный массив”), то можно использовать ассоциативный массив для экономии памяти.

Создание словаря

Пустой словарь можно создать при помощи функции dict() или пустой пары фигурных скобок {} (вот почему фигурные скобки нельзя использовать для создания пустого множества). Для создания словаря с некоторым набором начальных значений можно использовать следующие конструкции:

```
Capitals = {‘Poland': Warsaw', 'Ukraine': 'Kiev', 'USA': 'Washington'}
Capitals = dict(Poland = ‘Warsaw', Ukraine = 'Kiev', USA = 'Washington')
Capitals = dict([("Poland", "Warsaw"), ("Ukraine", "Kiev"), ("USA", "Washington")])
Capitals = dict(zip(["Poland", "Ukraine", "USA"], ["Warsaw", "Kiev", "Washington"]))
```

Первые два способа можно использовать только для создания небольших словарей, перечисляя все их элементы. Кроме того, во втором способе ключи передаются как именованные параметры функции dict, поэтому в этом случае ключи могут быть только строками, причем являющимися корректными идентификаторами. В третьем и четвертом случае можно создавать большие словари, если в качестве аргументов передавать уже готовые списки, которые могут быть получены не обязательно перечислением всех элементов, а любым другим способом построены по ходу исполнения программы. В третьем способе функции dict нужно передать список, каждый элемент которого является кортежем из двух элементов: ключа и значения. В четвертом способе используется функция zip, которой передается два списка одинаковой длины: список ключей и список значений.

Работа с элементами словаря

Основная операция: получение значения элемента по ключу, записывается так же, как и для списков: A[key]. Если элемента с заданным ключом не существует в словаре, то возникает исключение KeyError.

Другой способ определения значения по ключу — метод get: A.get(key). Если элемента с ключом get нет в словаре, то возвращается значение None. В форме записи с двумя аргументами A.get(key, val) метод возвращает значение val, если элемент с ключом key отсутствует в словаре.

Проверить принадлежность элемента словарю можно операциями in и not in, как и для множеств.

Для добавления нового элемента в словарь нужно просто присвоить ему какое-то значение: A[key] = value.

Для удаления элемента из словаря можно использовать операцию del A[key] (операция возбуждает исключение KeyError, если такого ключа в словаре нет. Вот два безопасных способа удаления элемента из словаря. Способ с предварительной проверкой наличия элемента:

```
if key in A:
    del A[key]
```

Способ с перехватыванием и обработкой исключения:

```
try:
    del A[key]
except KeyError:
    pass
```

Еще один способ удалить элемент из словаря: использование метода pop: A.pop(key). Этот метод возвращает значение удаляемого элемента, если элемент с данным ключом отсутствует в словаре, то возбуждается исключение. Если методу pop передать второй параметр, то если элемент в словаре отсутствует, то метод pop возвратит значение этого параметра. Это позволяет проще всего организовать безопасное удаление элемента из словаря: A.pop(key, None).

Перебор элементов словаря

Можно легко организовать перебор ключей всех элементов в словаре:

```
for key in A:
    print(key, A[key])
```

Следующие методы возвращают представления элементов словаря. Представления во многом похожи на множества, но они изменяются, если менять значения элементов словаря. Метод keys возвращает представление ключей всех элементов, метод values возвращает представление всех значений, а метод items возвращает представление всех пар (кортежей) из ключей и значений.

Соответственно, быстро проверить, если ли значение val среди всех значений элементов словаря A можно так: val in A.values(), а организовать цикл так, чтобы в переменной key был ключ элемента, а в переменной val было его значение можно так:

```
for key, val in A.items():
    print(key, val)
```

Кортеж

Кортежи служат для хранения нескольких объектов вместе. Их можно рассматривать как аналог списков, но без такой обширной функциональности, которую предоставляет класс списка. Одна из важнейших особенностей кортежей заключается в том, что они неизменяемы, так же, как и строки. Т.е. модифицировать кортежи невозможно.

Кортежи обозначаются указанием элементов, разделённых запятыми; по желанию их можно ещё заключить в круглые скобки.

Кортежи обычно используются в тех случаях, когда оператор или пользовательская функция должны наверняка знать, что набор значений, т.е. кортеж значений, не изменится.

Пример:

```
zoo = ('питон', 'слон', 'пингвин') # помните, что скобки не обязательны
print('Количество животных в зоопарке -', len(zoo))

new_zoo = 'обезьяна', 'верблюд', zoo
print('Количество клеток в зоопарке -', len(new_zoo))
print('Все животные в новом зоопарке:', new_zoo)
print('Животные, привезённые из старого зоопарка:', new_zoo[2])
print('Последнее животное, привезённое из старого зоопарка -', new_zoo[2][2])
print('Количество животных в новом зоопарке -', len(new_zoo)-1+len(new_zoo[2]))
```

Вывод:

Количество животных в зоопарке - 3
Количество клеток в зоопарке - 3
Все животные в новом зоопарке: ('обезьяна', 'верблюд', ('питон', 'слон', 'пингвин'))
Животные, привезённые из старого зоопарка: ('питон', 'слон', 'пингвин')
Последнее животное, привезённое из старого зоопарка - пингвин
Количество животных в новом зоопарке – 5

Как это работает:

Переменная zoo обозначает кортеж элементов. Как мы видим, функция len позволяет получить длину кортежа. Это также указывает на то, что кортеж является последовательностью.
Теперь мы перемещаем этих животных в новый зоопарк, поскольку старый зоопарк закрывается. Поэтому кортеж new_zoo содержит тех животных, которые уже там, наряду с привезёнными из старого зоопарка. Возвращаясь к реальности, обратите внимание на то, что кортеж внутри кортежа не теряет своей индивидуальности.

Доступ к элементам кортежа осуществляется указанием позиции элемента, заключённой в квадратные скобки – точно так же, как мы это делали для списков. Это называется оператором индексирования. Доступ к третьему элементу в new_zoo мы получаем, указывая new_zoo[2], а доступ к третьему элементу внутри третьего элемента в кортеже new_zoo – указывая new_zoo[2][2]. Это достаточно просто, как только вы поймёте принцип.

Скобки

Хотя скобки и не являются обязательными, я предпочитаю всегда указывать их, чтобы было очевидно, что это кортеж, особенно в двусмысленных случаях. Например, print(1,2,3) и print((1,2,3)) делают разные вещи: первое выражение выводит три числа, тогда как второе – кортеж, содержащий эти три числа.

Кортеж, содержащий 0 или 1 элемент

Пустой кортеж создаётся при помощи пустой пары скобок – “myempty = ()”. Однако, с кортежем из одного элемента не всё так просто. Его нужно указывать при помощи запятой после первого (и единственного) элемента, чтобы Python мог отличить кортеж от скобок, окружающих объект в выражении. Таким образом, чтобы получить кортеж, содержащий элемент 2, вам потребуется указать “singleton = (2,)”.

Строки

Строка – это последовательность символов. Чаще всего строки – это просто некоторые наборы слов

Строка считывается со стандартного ввода функцией input(). Напомним, что для двух строк определа операция сложения (конкатенации), также определена операция умножения строки на число.

Строка состоит из последовательности символов. Узнать количество символов (длину строки) можно при помощи функции len:

```
>>> S = 'Hello'
>>> print(len(S))
5
```

Основные операции над строками:

A + B — конкатенация;
A * n — повторение n раз, значение n должно быть целого типа.

Преобразование типов

Иногда бывает полезно целое число записать, как строку. И, наоборот, если строка состоит из цифр, то полезно эту строку представить в виде числа, чтобы дальше можно было выполнять арифметические операции с ней. Для этого используются функции, одноименные с именем типа, то есть int, float, str. Например, int('123') вернет целое число 123, а str(123) вернет строку '123'.

Пример:

```
>>> str(2 + 2) * int('2' + '2')
'44444444444444444444'
```

Результатом будет строка из числа 4, повторенная 22 раза.

Лекцию закончено.
Благодарю за внимание!