КАЛУЖСКИЙ ФИЛИАЛ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО
ОБРАЗОВАТЕЛЬНОГО УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ Н.Э. БАУМАНА
(национальный исследовательский университет)»

Факультет «Информатика и управление»

Кафедра «Программное обеспечение ЭВМ, информационные технологии»

Высокоуровневое программирование

Лекция №9-10. «Функции Python»

Калуга - 2020

Функции

- Функция - это блок кода, выполняющий определенные действия:
  - у функции есть имя, с помощью которого можно запускать этот блок кода сколько угодно раз
  - запуск кода функции называется вызовом функции
  - при создании функции, как правило, определяются параметры функции
  - параметры функции определяют, какие аргументы функция может принимать
  - функциям можно передавать аргументы
  - соответственно, код функции будет выполняться с учетом указанных аргументов

Сегментация данных

- **Статическая память** (выделяется до начала исполнения программы)
- **Динамическая память** (куча, heap – выделяется по запросу программиста)
- **Автоматическая память** (стековая - создание объектов автоматической памяти совершается компилятором, и компилятором эти же объекты разрушаются)

Механизм работы

- Большинство современных языков программирования для управления вызовом подпрограмм используют ***стек вызовов***.
- Примерный цикл работы стека вызова следующий:
  - Вызов подпрограммы создает запись в стеке; каждая запись может содержать информацию о данных вызова (аргументах, результате, а также адресе возврата).
  - Когда подпрограмма завершается, запись удаляется из стека и программа продолжает выполняться, начиная с адреса возврата.

Пример стека вызовов

```
def g():
    print('Inside g')

def f():
    print('Inside f')
    g()
    print('Inside f')

print('1. Inside module')
f()
print('2. Inside module')
```

```
1. Inside module
Inside f
Inside g
Inside f
2. Inside module
```

Пример получения стека через модуль *traceback* и его печати

```
32  import traceback
33
34  def f():
35      g()
36
37  def g():
38      for line in traceback.format_stack():
39          print(line.strip())
40
41  f()
```

```
File "C:/Projects/Py/SecondCourse/main.py", line 41, in <module>
    f()
File "C:/Projects/Py/SecondCourse/main.py", line 35, in f
    g()
File "C:/Projects/Py/SecondCourse/main.py", line 38, in g
    for line in traceback.format_stack():
```

Функции в Python

- функции создаются с помощью зарезервированного слова ***def***
- за ***def*** следуют имя функции и круглые скобки
- внутри скобок могут указываться параметры, которые функция принимает
- после круглых скобок идет двоеточие и с новой строки, с отступом, идет блок кода, который выполняет функция
- первой строкой, опционально, может быть комментарий, так называемая ***docstring***

Оператор *return*

- в функциях может использоваться оператор ***return***
- он используется для прекращения работы функции и выхода из неё
- чаще всего, оператор ***return*** возвращает какое-то значение
- в Python все данные - объекты, при вызове в функцию передается ссылка на этот объект
- при этом, изменяемые объекты передаются по ссылке, неизменяемы – по значению.

Преимущества и недостатки

Главное назначение подпрограмм сегодня - структуризация программы с целью удобства ее понимания и сопровождения.

Преимущества использования подпрограмм:

- декомпозиция сложной задачи на несколько более простых подзадач: это один из двух главных инструментов структурного программирование (второй - структуры данных);
- уменьшение дублирования кода и возможность повторного использования кода в нескольких программах - следование принципу DRY «не повторяйся» (англ. Don’t Repeat Yourself);
- распределение большой задачи между несколькими разработчиками или стадиями проекта;
- сокрытие деталей реализации от пользователей подпрограммы;
- улучшение отслеживания выполнения кода (большинство языков программирования предоставляет стек вызовов подпрограмм).

Недостатком использования подпрограмм можно считать накладные расходы на вызов подпрограммы, однако современные трансляторы стремятся оптимизировать данный процесс.

Виды функций

- Глобальные
  - Доступны из любой точки программного кода в том же модуле или из других модулей.
- Локальные (вложенные)
  - Объявляются внутри других функций и видны только внутри них: используются для создания вспомогательных функций, которые нигде больше не используются.
- Анонимные
  - Не имеют имени и объявляются в месте использования. В Python они представлены лямбда-выражениями.
- Методы
  - Функции, ассоциированные с каким-либо объектом (например, list.append(), где append() - метод объекта list).

Параметры и аргументы

Все параметры, указываемые в Python при объявлении и вызове функции делятся на:

- позиционные: указываются простым перечислением:

```
def function_name(a, b, c):  # a, b, c - 3 позиционных параметра
    pass
```

- ключевые: указываются перечислением ключ=значение:

```
def function_name(key=value, key2=value2):  # key, key2 - 2 позиционных аргумента
    pass                                    # value, value2 - их значения по умолчанию
```

- Позиционные и ключевые аргументы могут быть скомбинированы. Синтаксис объявления и вызова функции зависит от типа параметра, однако позиционные параметры (и соответствующие аргументы) всегда идут перед ключевыми

Объявление функции

```
def example_func(a, b, c):          # можно : 'a', 'b', 'c' - позиционные параметры
    pass

def example_func(a, b, c=3):        # можно : 'a', 'b' - позиционные параметры,
    pass                            #         'c' - ключевой параметр

def example_func(a=1, b=2, c=3):    # можно : 'a', 'b', 'c' - ключевые параметры
    pass

def example_func(a=1, c, b=2):      # нельзя: ключевой параметр 'a'
    pass                            #         идет раньше позиционнных
```

```python
def example_func(a, b, c=3):  # a, b - позиционные параметры, c - ключевой параметр
    pass

# Вызовы функции
example_func(1, 2, 5)          # можно : аргументы 1, 2, 5 распределяются
                               #         позиционно по параметрам 'a', 'b', 'c'

example_func(1, 2)             # можно : аргументы 1, 2 распределяются позиционно
                               #         по параметрам 'a', 'b'
                               #         в ключевой параметр 'c' аргумент
                               #         не передается, используется значение 3

example_func(a=1, b=2)         # можно : аналогично example_func(1, 2),
                               #         все аргументы передаются по ключу

example_func(b=2, a=1)         # можно : аналогично example_func(a=1, b=2),
                               #         если все позиционные параметры заполнены как
                               #         ключевые аргументы, можно не соблюдать порядок

example_func(c=5, b=2, a=1)    # можно : аналогично example_func(1, 2),
                               #         аргументы передаются по ключу

example_func(1)                # нельзя: для позиционного аргумента 'b'
                               #         не передается аргумент

example_func(b=1)              # нельзя: для позиционного аргумента 'a'
                               #         не передается аргумент
```

Преимущества ключевых параметров

- нет необходимости отслеживать порядок аргументов;
- у ключевых параметров есть значение по умолчанию, которое можно не передавать.

Упаковка и распаковка аргументов

В ряде случаев бывает полезно определить функцию, способную принимать любое число аргументов. Так, например, работает функция print(), которая может принимать на печать различное количество объектов и выводить их на экран.

Достичь такого поведения можно, используя механизм упаковки аргументов, указав при объявлении параметра в функции один из двух символов:

- *: все позиционные аргументы начиная с этой позиции и до конца будут собраны в кортеж;
- **: все ключевые аргументы начиная с этой позиции и до конца будут собраны в словарь.

Упаковка аргументов

```
def print_arguments(* some_tuple, ** some_dict):
    i = 1
    for item in some_tuple:
        print(f'{i} - {item}')
        i += 1


    for key, value in some_dict.items():
        print(f'{key}:  {value}')

print_arguments('Иванов', 'Петров', 'Сидоров', group='ITD-32', year=2020)
```

```
1 - Иванов
2 - Петров
3 - Сидоров
group:  ITD-32
year:  2020
```

Распаковка аргументов

Python также предусматривает и обратный механизм - распаковку аргументов, используя аналогичные обозначения перед аргументом:

- *: кортеж/список распаковывается как отдельные позиционные аргументы и передается в функцию;

- **: словарь распаковывается как набор ключевых аргументов и передается в функцию

Распаковка аргументов

```
def summ(*elements):
    result = 0
    for i in elements:
        result += i
    return result

print(f'sum = {summ(1, 2, 3, 4, 5)}')
print(f'sum = {summ(1, 2, 3, 4, 5, 10)}')
mas = [0, 2, 4, 6]
print(f'sum = {summ(*mas)}')
#такой вызов эквивалентен вызову summ(0, 2, 4, 6)
```

```
sum = 15
sum = 25
sum = 12
```

Что такое *args и **kwargs в Python?

```
1  def print_numbers(a,b,c):
2      print(a, "is stored in a")
3      print(b, "is stored in b")
4      print(c, "is stored in c")
5
6  print_numbers(1,2)
```

```
---------------------------------------
TypeError
<ipython-input-1-46879d054182> in <mod
      3      print(b, "is stored in b")
      4      print(c, "is stored in
----> 5 print_numbers(1,2)

TypeError: print_numbers() missing
```

```
1  def print_numbers(a,b,c=None):
2      print(a, "is stored in a")
3      print(b, "is stored in b")
4      print(c, "is stored in c")
5
6  print_numbers(1,2)
```

```
1 is stored in a
2 is stored in b
None is stored in c
```

```
1  def print_numbers(a=None,b=None,c=None):
2      print(a, "is stored in a")
3      print(b, "is stored in b")
4      print(c, "is stored in c")
5
6  print_numbers(c=3, a=1)
```

```
1 is stored in a
None is stored in b
3 is stored in c
```

Что такое *args и **kwargs в Python?

- Оператор * позволяет «распаковывать» объекты, внутри которых хранятся некие элементы.

```
1 a = [1,2,3]
2 b = [*a,4,5,6]
3 print(b)
```

```
[1, 2, 3, 4, 5, 6]
```

- В этом примере берётся содержимое списка a, распаковывается, и помещается в список b.

Что такое *args и **kwargs в Python?

- ***args** – это сокращение от «arguments» (аргументы)
- ****kwargs** – сокращение от «keyword arguments» (именовнные аргументы)

Каждая из этих конструкций используется для распаковки аргументов соответствующего типа, позволяя вызывать функции со списком аргументов переменной длины.

Что такое *args и **kwargs в Python?

```python
def print_exam_scores(student, *scores):
    print(f"Student Name: {student}")
    for score in scores:
        print(score)

print_exam_scores("Иванов", 100, 95, 88)
print_exam_scores("Петров", 65, 80, 91, 88)
print_exam_scores("Сидоров", 89, 93, 90, 100, 75)
```

```
Student Name: Иванов
100
95
88
Student Name: Петров
65
80
91
88
Student Name: Сидоров
89
93
90
100
75
```

- Нет ли тут ошибки? Ошибки здесь нет.
- Дело в том, что «args» – это всего лишь набор символов, которым принято обозначать аргументы.
- Самое главное тут – это оператор *.

Что такое *args и **kwargs в Python?

```python
def print_faculty_students(department, **students):
    print(f"Name of department: {department}")
    for group, name in students.items():
        print(f"{group}: {name}")

print_faculty_students("ИУК4", group="ИУК4-31Б", names=["Иванов", "Петроа", "Сидоров"])
print_faculty_students("ИУК4", group="ИУК4-32Б", names=["Герасимов", "Федоренко", "Николаев"])
print_faculty_students("ИУК5", group="ИУК5-11Б", names=["Сидоренко", "Антонова", "Носова"])
print_faculty_students("ИУК5", group="ИУК5-12Б", names=["Иванова", "Хохлова", "Ефремов"])
```

```
Name of department: ИУК4
group: ИУК4-31Б
names: ['Иванов', 'Петроа', 'Сидоров']
Name of department: ИУК4
group: ИУК4-32Б
names: ['Герасимов', 'Федоренко', 'Николаев']
Name of department: ИУК5
group: ИУК5-11Б
names: ['Сидоренко', 'Антонова', 'Носова']
Name of department: ИУК5
group: ИУК5-12Б
names: ['Иванова', 'Хохлова', 'Ефремов']
```

Что такое *args и **kwargs в Python?
Выводы

- Используйте общепринятые конструкции *args и **kwargs для захвата позиционных и именованных аргументов.
- Конструкцию **kwargs нельзя располагать до *args. Если это сделать — будет выдано сообщение об ошибке.
- Остерегайтесь конфликтов между именованными параметрами и **kwargs, в случаях, когда значение планируется передать как **kwarg-аргумент, но имя ключа этого значения совпадает с именем именованного параметра.
- Оператор *можно использовать не только в объявлениях функций, но и при их вызове.

Область видимости

- ***Область видимости*** - область программы, где определяются идентификаторы, и транслятор выполняет их поиск. За пределами области видимости тот же самый идентификатор может быть связан с другой переменной, либо быть свободным (не связанным ни с какой из них).

Область видимости

В Python выделяется четыре области видимости:

- **Локальная** (англ. *Local*)
  - Собственная область внутри инструкции def.
- **Нелокальная** (англ. *Enclosed*)
  - Область в пределах вышестоящей инструкции def.
- **Глобальная** (англ. *Global*)
  - Область за пределами всех инструкций def - глобальная для всего модуля.
- **Встроенная** (англ. *Built-in*).
  - «Системная» область модуля builtins: содержит предопределенные идентификаторы, например, функцию max() и т.п.

Основные положения

- идентификатор может называться локальным, глобальным и т.д., если имеет соответствующую область видимости;

- функции образуют локальную область видимости, а модули – глобальную;

- чем ближе область к концу списка, тем более она открыта (ее содержимое доступно для более закрытых областей видимости; например, глобальные идентификаторы и предопределенные имена могут быть доступны в локальной области видимости функции, но не наоборот).

Основные положения

- Схема разрешения имен в языке Python называется правилом LEGB (Local, Enclosed, Global, Built-in): когда внутри функции выполняется обращение к неизвестному имени, интерпретатор пытается отыскать его в четырех областях видимости по очереди до первого нахождения.

Возврат нескольких значений

- Часто из функции необходимо вернуть несколько значений (например, в Паскале, для этого используются выходные параметры с ключевым словом ***var***). Одним из лучших способов для этого в Python является возврат кортежа с несколькими значениями.

```python
def solve_equation(a, b, c):
    d = b**2 - 4 * a * c

    if d < 0:
        return(0, ())

    if d == 0:
        x = -b / (2*a)
        return(1, (x,))
    else:
        x1 = (-b - d**0.5) / (2*a)
        x2 = (-b + d**0.5) / (2*a)
        return (2, (x1, x2))

a = int(input())
b = int(input())
c = int(input())

num, x = solve_equation(a, b, c)
if num == 0:
    print("решений нет")
elif num == 1:
    print("x = ", x[0])
else:
    print(f"x1 = {x[0]}, x2 = {x[1]}")
```

```
1
4
3
x1 = -3.0, x2 = -1.0

1
2
3
решений нет


5
0
0
x =  0.0
```

Рекурсия

- Рекурсия - вызов функции внутри самой себя, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия)
- Количество вложенных вызовов функции или процедуры называется глубиной рекурсии. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причем без явных повторений частей программы и использования циклов.
- Не рекомендуется использовать рекурсию, если такая функция может привести или приводит к большой глубине рекурсии - лучше заменить ее циклической конструкцией. Рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека (англ. англ. Stack Overflow) вызовов.

Рекурсия

```python
def factorial_cycle(x):
    res = 1
    for i in range(x):
        res *= i + 1
    return res

def factorial_rec(x):
    if x == 1:
        return 1
    else:
        return x * factorial_rec(x - 1)

print(factorial_cycle(5))   # 120
print(factorial_rec(5))     # 120
```

Анонимные функции

- Анонимные функции это однострочные функции, которые используются в случаях, когда вам не нужно повторно использовать функцию в программе. Они идентичны обыкновенным функциям и повторяют их поведение.

- **Образец**

```
lambda argument: manipulate(argument)
```

Анонимные функции (пример)

```
1 def f(x, y, z):
2     return x + y + z
3
4 f(3, 5, 7)
```

```
15
```

```
1 add = lambda x, y, z: x + y + z
2
3 print(add(3, 5, 7))
```

```
15
```

- Обратите внимание, что в определении лямбда-функции нет оператора **return**, так как в этой функции может быть только одно выражение, которое всегда возвращает значение и завершает работу функции.

Различия lambda от def

- lambda – это выражение, а не инструкция. По этой причине ключевое слово lambda может появляться там, где синтаксис языка Python не позволяет использовать инструкцию def, – внутри литералов или в вызовах функций, например.
- Тело lambda – это не блок инструкций, а единственное выражение. Тело lambda-выражения сродни тому, что вы помещаете в инструкцию return внутри определения def, – вы просто вводите результат в виде выражения вместо его явного возврата.

Анонимные функции (примеры)

- Использование значений по умолчанию в lambda-выражениях:

```
1 x = (lambda a='a', b='b', c='c': a + b + c)
2 x('x')
```

```
'xbc'
```

- Реализация логики выбора внутри lambda-функций:

```
1 lower = (lambda x, y: x if x < y else y)
2 print(lower('bb', 'aa'))
3 print(lower('aa', 'bb'))
```

```
aa
aa
```

Анонимные функции (примеры)

- Сортировка списка:

```
1  a = [(1, 2), (4, 1), (9, 10), (13, -3)]
2  a.sort(key=lambda x: x[1])
3
4  print(a)
```

```
[(13, -3), (4, 1), (1, 2), (9, 10)]
```

```
1  b = [12, 23, 41, 45, 67, 98]
2  b.sort(key=lambda n: n % 10)
3  b
```

```
[41, 12, 23, 45, 67, 98]
```

Анонимные функции (примеры)

- Создание таблиц переходов, которые представляют собой списки или словари действий, выполняемых по требованию:

```
1  L = [lambda x: x**2,    # Встроенные определения функций
2       lambda x: x**3,
3       lambda x: x**4]    # Список из трех функций
4
5  print(L)
6
7  for f in L:
8      print(f(2))         # Выведет 4, 8, 16
9
10 print(L[0](3))          # Выведет 9
```

```
[<function <lambda> at 0x060DC808>, <function <lambda> at 0x06
4
8
16
9
```

Анонимные функции (примеры)

- Создание таблиц переходов, которые представляют собой списки или словари действий, выполняемых по требованию:

```
1  def f1(x): return x ** 2
2  def f2(x): return x ** 3    # Определения именованных функций
3  def f3(x): return x ** 4
4
5  L = [f1, f2, f3]            # Ссылка по имени
6
7  for f in L:
8      print(f(2))             # Выведет 4, 8, 16
9
10 print(L[0](3))              # Выведет 9
```

```
4
8
16
9
```

Задачи для самостоятельного решения

- Даны n предложений. Определите, сколько из них содержат хотя бы одну цифру.
- Дана строка s и символ k. Реализуйте функцию, рисующую рамку из символа k вокруг данной строки, например:

```
**************
*Текст в рамке*
**************
```

- Для введенного предложения выведите статистику *символ=количество*. Регистр букв не учитывается.
- Дата характеризуется тремя натуральными числами: день, месяц и год. Учитывая, что год может быть високосным, реализуйте две функции, которые определяют вчерашнюю и завтрашнюю дату.

Задачи для самостоятельного решения

- Напишите функцию, которая принимает неограниченное количество числовых аргументов и возвращает кортеж из двух списков:
  - отрицательных значений (отсортирован по убыванию);
  - неотрицательных значений (отсортирован по возрастанию).
- Составьте две функции для возведения числа в степень: один из вариантов реализуйте в рекурсивном стиле.
- Дано натуральное число. Напишите рекурсивные функции для определения:
  - суммы цифр числа;
  - количества цифр в числе.