Урок 3

Функции и работа с файлами

Функции как строительные блоки программ. Встроенные функции. Области видимости и lambda-функции. Работа с файлами. Сохранение python-объектов в файл. Формат JSON.

Встроенные функции (часть 1)

Встроенные функции решают наиболее часто возникающие задачи. Мы уже пользовались некоторыми встроенными функциями python, такими как:

- print();
- input();
- len();
- функциями преобразования типов int(), float(), bool() и т.д.

Кратко рассмотрим ещё группу наиболее используемых функций:

- range([start=0], stop, [step=1]) — арифметическая прогрессия от start до stop с шагом step;
- abs(x) — возвращает абсолютную величину (модуль числа);
- max(iter, [args ...] * [, key]) — максимальный элемент последовательности. min();
- round(X [, N]) — округление до N знаков после запятой;
- sum(iter, start=0) — сумма членов последовательности;
- type(object) — возвращает тип объекта;
- enumerate(string) — возвращает пары, (элемент, его индекс)

Если принцип какой-то функции неочевиден из описания — поэкспериментируйте в оболочке интерпретатора или посмотрите примеры в проекте.

Обратите внимание! Функции max(), min(). sum(). len() принимают в качестве первого аргумента объект-итератор (любую последовательность).

Функции

Понятно, что для всех ситуаций нельзя предусмотреть готовые функции. Пришло время научиться писать свои собственные.

Функции в программировании можно представить как изолированный блок кода, обращение к которому в процессе выполнения программы может быть многократным. Зачем нужны такие блоки инструкций? В первую очередь, чтобы сократить объём исходного кода: рационально вынести часто повторяющиеся выражения в отдельный блок и затем, по мере надобности, обращаться к нему.

Определим простейшую функцию:

```
# определение функции
def summ(a, b):
    c = a + b
    return c
# вызов функции
res = summ(5, 10)
print(res)
```

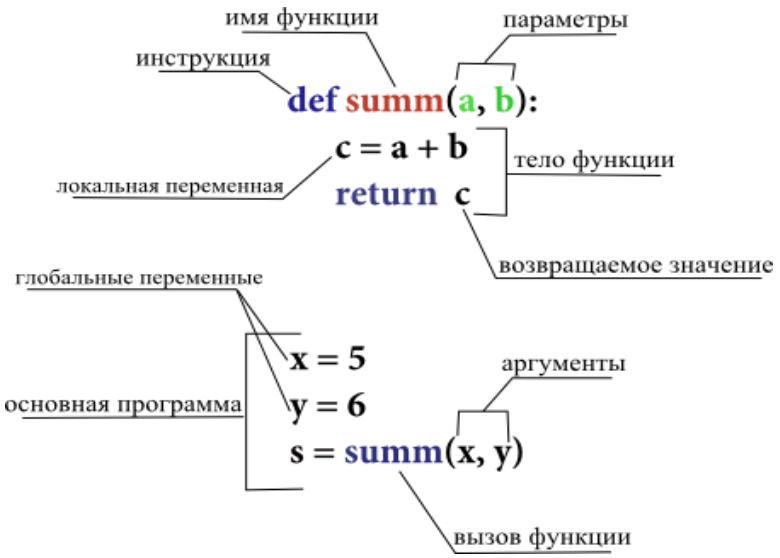

Рис. Схема и термины функции

def — это инструкция (команда) языка программирования Python, позволяющая создавать функцию.

add — это имя функции, которое (как и имена переменных) может быть почти любым, но, желательно, осмысленным. К имени функции применяются те же требования, что и к имени переменной.

После в скобках перечисляются параметры функции. Если их нет, то скобки остаются пустыми. Далее идёт двоеточие, обозначающее окончание заголовка функции (аналогично с условиями и циклами).

После заголовка с новой строки и с отступом следуют выражения тела функции. В конце тела функции присутствует инструкция return (может и не быть), которая возвращает значение в основную ветку программы. В данном случае, если бы в функции не было инструкции return, то в основную программу ничего бы не возвращалось[1].

Выражения тела функции выполняются лишь тогда, когда она вызывается в основной программе. Так, если функция присутствует в исходном коде, но нигде не вызывается в нём, то содержащиеся в ней инструкции не будут выполнены ни разу.

По сути функция — это своего рода «черный ящик». После того как тело функции написано и функция работает правильно, вам неважно, как именно работает её содержимое, главное — передать аргументы и получить результат (возвращаемое значение). До изучения функций вы уже не раз ими пользовались, например, функция для ввода input() и функция для вывода print(). Вас хоть раз интересовало, как эти функции устроены внутри?

Также функции позволяют разбить программу на логические части для упрощения тестирования.

Документирование функций

Если вам нужно изучить код функции, чтобы начать ею пользоваться, это явно плохая функция. Когда вы создаёте функции — вы создаёте инструменты для повторного использования. Вряд ли через год вы будете помнить, какая из ваших функций что делает. Чтобы избежать таких проблем, рекомендуется писать к функциям краткие комментарии. Комментарии принято добавлять в виде многострочной строки после заголовка функции.

[1] На самом деле, если функция ничего не возвращает, она возвращает специальный тип None.

```
def summ(a, b):
    """
    Возвращает сумму аргументов
    """
    c = a + b
    return c
```

Аргументы функции

Чтобы функция выполнила определённый алгоритм, ей нужны данные. Данные передаются функции при её вызове в скобках и называются аргументами. Однако, чтобы функция могла «взять» передаваемые ей данные, необходимо при её создании описать параметры (в скобках после имени функции), представляющие собой переменные.

Когда функция вызывается, конкретные аргументы подставляются вместо параметров-переменных. Почти всегда количество аргументов и параметров должно совпадать (хотя можно запрограммировать переменное количество принимаемых аргументов, но об этом позже). В качестве аргументов могут выступать как непосредственно значения, так и переменные, ссылающиеся на них.

```
def average(a,b,c,d):
    sum = a + b + c + d
    return sum/4
a = 8
b = 4
c = -12
result = average(a,c,b,10)
print(result)
```

В качестве аргументов в функцию arithmetic_mean() передаются значения переменных a.b.c и число 10. А переменные-параметры a,b,c,d принимают эти значения.

Глобальные и локальные переменные

Все переменные, созданные внутри функции, а также переменные-параметры (указанные в скобках после имени функции) являются **локальными** и существуют только во время выполнения функции.

Глобальные переменные — переменные, объявленные в основной части программы, т.е. вне функции. Глобальные переменные существуют до конца времени выполнения программы.

Технически функции могут пользоваться глобальными переменными, но делать это не рекомендуется.

```
z = 15
def local(x):
    # x, y - локальные переменные, доступны только во время выполнения функции
    # z - глобальная переменная, доступна во всем модуле (.py файле)
    y = 10
    print('x = {}, y = {}, z = {}'.format(x, y, z))
local(5)
```

Чем плохо использование глобальных переменных в функции? Функции — это инструменты, выполняющие определённые подзадачи (мини-алгоритмы). По идее, нужно стремиться к переносимости функции в любую другую программу, а если функция использует глобальные переменные, то она становится привязана к ним, что затрудняет переносимость.

Функция как объект

Когда интерпретатор встречает инструкцию def test, создаётся переменная test с указателем на объект-функцию, поэтому функции можно объявлять внутри других инструкций и даже в других функциях.

```
access = True
if access:
    def render(name):
        return 'Welcome, %s' % name
else:
    def render(name):
        return '%s, sorry. Try to enter again.' % name

print(render('Иван'))
```

А также возможно передавать ссылку на функцию внутрь другой функции.

```
def mult(x):
    return x**2
# Переменная a - ссылка на функцию
a = mult
# Используя эту переменную, можно вызвать функцию
print(a(4))  # тоже, что и  print(mult(4))
```

lambda-функции

Анонимные (lambda) функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции lambda. Кроме этого, их необязательно присваивать переменной.

```
# В переменную f будет сохранена ссылка на объект-функцию
f = lambda x: x ** 2
# Суть в том, что инструкция lambda возвращает ссылку на функцию
# Тоже, что и
# def mult(x):
#    return x**2
# Только объявляется компактнее и работает быстрее
print(f(4))
# lambda-функции можно использовать и в выражении
print((lambda x: x ** 2)(4))
```

Область видимости

Под термином «область видимости» подразумевается пространство имён, то есть место в программном коде, где имени было присвоено значение и где это имя доступно в программе.

Практически всё, что имеет отношение к именам, включая классификацию областей видимости, в языке Python связано с операциями присваивания. Как мы уже видели, имена появляются в тот момент, когда им впервые присваиваются некоторые значения, прежде чем имена смогут быть использованы, им необходимо присвоить значения. Поскольку имена не объявляются заранее, интерпретатор Python по местоположению операции присваивания связывает имя с конкретным пространством имен. Другими словами, место, где выполняется присваивание, определяет пространство имён, в котором будет находиться имя, следовательно, и область его видимости.

Изолируемые области видимости переменных в python создают только функции.

В python есть 4 области видимости:

1. Локальная.
2. Объемлющей функции.
3. Глобальная(модуля).
4. Встроенная (builtins) — предопределённые имена, например, имена встроенных функций.

Поиск переменной происходит поочередно с 1 по 4-ую.

```
x = 5               # глобальная переменная - доступна в любом месте данного модуля(файла)
def outside():
    y = 10      # доступна в теле данной функции + во всех вложенных
    def inside():
        z = 15 # доступна только в теле данной функции
        print('inside x: {}, y: {}, z: {}'.format(x, y, z))
    inside()
    print('outside x: {}, y: {}, z: {}'.format(x, y, 'z недоступна'))
outside()
print('inside x: {}, y: {}, z: {}'.format(x, 'y недоступна', 'z недоступна'))
x = 5
def wrapper():
    def test1():
        x = 10 # локальная переменная x перекрывает видимость глобальной x
        print('test1 x = ', x)
    def test2():
        print('test2 x = ', x)
# x = 22 # ^-- ошибка, выше используем переменную, объявленную позднее
    def test3():
        global x # инструкция global - поиск переменной в глобальной области
# Есть инструкция nonlocal -  поиск переменной в объемлющей функции
        print('test3 x = ', x)
        x = 25
    test1()
    test2()
    test3()
wrapper()
print('after wrapper x = ', x)
```

Произвольное количество аргументов

Для получения неопределённого (любого) количества аргументов используют конструкцию: *args в качестве параметра функции, где args — произвольное имя.

Используя эту особенность, мы можем написать более универсальную функцию вычисления среднего арифметического, вместо функции, работающей только с 4-мя аргументами.

```
def average(*args):
    summ = 0
    for arg in args:
        summ += arg
return summ/len(args)
```

Кортеж, элементами которого будут переданные значения аргументов, — это args.

Т.е., если мы вызовем функцию average(1,2,3,4,5), то в теле функции будет создан кортеж:

args=(1,2,3,4,5)

Именованные аргументы

Если нам нужно передать функции много аргументов, то довольно сложно не запутаться, в каком порядке их передавать, ведь если перепутать порядок, не то значение будет передано не в ту переменную. В python существуют именованные аргументы.

```
def print_info(**kwargs):
      print("You name is %s %s. You age is %s. And your address is: %s"%
      (kwargs["name"],kwargs["surname"],kwargs["age"],kwargs["adress"]))
print_info(name="Василий",surname="Иванов",age="12",adress="ул.Белана 22")
```

Чтобы принять именованные аргументы используется **kwargs в качестве имени параметра функции. В теле функции kwargs — обычный словарь, в котором ключами являются имена аргументов.

Для данного примера в теле функции словарик выгляди так:

```
{'adress': 'ул.Белана 22', 'age': '12', 'surname': 'Иванов', 'name': 'Василий'}
```

Значения по умолчанию

Давайте создадим свою функцию с параметром по умолчанию:

```
def welcome(name="Инкогнито"):
    print("Приветствую вас, %s"%(name))
```

Попробуем вызвать функцию с параметром и без:

```
>>> welcome("User")
: Приветствую вас, User
>>> welcome()
: Приветствую вас, Инкогнито
```

Параметры по умолчанию позволяют делать функции удобнее в использовании.

Рассмотрим подробнее функцию print(), которой пользовались уже много раз.

Чтобы посмотреть объявление функции, наберите print() (в PyCharm-е) и ctrl+ЛКМ по имени функции. Откроется файл builtins.py, и там будет нужное нам объявление:

```
def print(*args, sep=' ', end='\n', file=None)
```

Параметры sep, end и file как раз и являются параметрами по умолчанию. Т.е. если вы не указываете значения этих параметров, то в теле функции используются те значении, которые заданы в объявлении, но вы можете указать значения явно.

Для начала рассмотрим, за что отвечают значения этих параметров:

- sep — строка, вставляемая между выводимыми значениями;
- end — строка, вставляемая в конец вывода функции print();
- file — файл потока вывода (рассмотрим позднее).

Чтобы все стало понятным, наберите код:

```
>>> print("Иван","Иванович","Иванов",sep="//",end="!!!")
```

Получите: Иван//Иванович//Иванов!!!

Поэкспериментируйте с другими значениями параметров по умолчанию.

Теперь вы можете использовать значения по умолчанию функции print() для своего удобства.

Встроенные функции (часть 2)

Рассмотрим еще несколько встроенных функций. Только после изучения изложенного материала эти функции станут понятными.

zip()

Принцип работы этой функции проще показать на примерах.

```
a = [1,2]
b = [3,4]
print(zip(a,b))
```

Выведет: [(1, 3), (2, 4)]

```
a = [1,2,4]
b = [3,4]
c = [5,6,0]
print zip(a,b,c)
```

Выведет:[(1, 3, 5), (2, 4, 6)]

Берёт по минимальному количеству элементов, остальные будут отброшены.

map()

Позволяет применить функцию к каждому элементу последовательности, результаты функции возвращает в виде итератора.

Например, нам нужно возвести каждый элемент последовательности в квадрат:

```
print(list(map(lambda x: x*x, [2, 5, 12, -2])))
```

map(func_link, <итератор>) --> итератор, каждым элементом которого является применение функции funk_link к элементам исходного итератора.

Результат оборачиваем в list(), чтобы увидеть полный результат.

filter()

```
#  filter(filter_func, <итератор>)  -->  итератор  с  отфильтровыванием  элементов
функцией filter_func
print(list(filter(lambda x: x > 5, [2, 10, -10, 8, 2, 0, 14])))
# Отбрасываем все элементы долиной НОЛЬ
print(list(filter(len, ['', 'not null', 'bla', '', '10'])))
```

T.e. filter() отбрасывает те элементы, для которых функция возвращает False.

Работа с файлами

Прежде чем работать с файлом, его надо открыть. С этим замечательно справится встроенная функция open().

```
import os
# не самый хороший способ задания пути:
path = 'files/text.txt'
# хороший кроссплатформенный метод указания пути:
path = os.path.join('files', 'text.txt')
f = open(path, 'r', encoding='UTF-8')
# Считываем всю информацию из файла в виде списка строк
print(f.readlines())
f.close()
```

encoding='UTF-8' — указываем кодировку файла. Без этой строки под Windows много проблем.

‘r’ — режим «на чтение».

Режимы работы с файлом:

Режим	**Обозначение**
‘r’	открытие на чтение (является значением по умолчанию)
‘w’	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый
‘x’	открытие на запись, если файла не существует, иначе исключение
‘a’	открытие на дозапись, информация добавляется в конец файла
‘b’	открытие в двоичном режиме
‘+’	открытие на чтение и запись

Режимы могут быть объединены, то есть, к примеру, 'rb' — чтение в двоичном режиме. По умолчанию режим равен 'rt'.

При открытии файла open() возвращает итератор, используя цикл for in, мы можем читать информацию построчно.

```
path = os.path.join('files', 'text.txt')
f = open(path, 'r', encoding='UTF-8')
wanted_symbol = "+"
for line in f:                  # считываем файл построчно
    if wanted_symbol in line:   # пока не найдем нужную информацию
        print(line)
        break                   # как нашли, заканчиваем чтение файла
```

Рекомендуется работать с файлами, используя менеджер контекста with.

```
# Наиболее правильный способ работы с файлами
# По окончании инструкции with, файл гарантировано будет закрыт, даже если произойдёт ошибка
with open(path, 'r', encoding='UTF-8') as f:
    print(f.readlines())
```

Для записи информации в файл используется метод .write().

```
my_file = open("some.txt", "w")
my_file.write("Мне нравится Python!\Это классный язык!")
my_file.close()
```

Домашнее задание

1. Смотреть здесь https://github.com/GeekBrainsTutorial/Python_lessons_basic/tree/master/lesson03.

Большинство заданий делятся на три категории — easy, normal и hard:

- easy — простенькие задачи на понимание основ;
- normal — если вы делаете эти задачи, то вы хорошо усвоили урок;
- hard — наиболее хитрые задачи, часто с подвохами, для продвинутых слушателей.

Если вы не можете сделать normal задачи — это повод пересмотреть урок, перечитать методичку и обратиться к преподавателю за помощью.

Если не можете сделать hard — не переживайте, со временем научитесь.

Решение большинства задач будем разбирать в начале каждого вебинара.

Дополнительные материалы

Всё то, о чём сказано здесь, но подробнее:

1. Функции и аргументы.
2. Функции map, zip, lambda.
3. Работа с файлами.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Учим python качественно(habr).
2. Самоучитель по python.
3. Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание).
4. younglinux.