Лабораторная работа 2.11 Замыкания в языке Python

Цель работы: *приобретение навыков по работе с замыканиями при написании программ с помощью языка программирования Python версии 3.x.*

Ход работы

Что такое замыкание?

Для начала обратимся к википедии: “замыкание (*closure*) в программировании — это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся ее параметрами.” Перед тем как перейти к рассмотрению примеров реализации замыканий на *Python*, для начал вспомним тему “область видимости переменных”. Обычно, по области видимости, переменные делят на глобальные и локальные. Глобальные существует в течении всего времени выполнения программы, а локальные создаются внутри методов, функций и прочих блоках кода, при этом, после выхода из такого блока переменная удаляется из памяти.

Что касается *Python*, то тут выделяют четыре области видимости для переменных (с вашего позволения я буду использовать английские термины).

- **Область видимости** *Local*

Эту область видимости имеют переменные, которые создаются и используются внутри функций.

Пример.

```
>>> def add_two(a):
        x = 2
        return a + x

>>> add_two(3)
5

>>> print(x)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

В данной программе объявлена функция *add_two()*, которая прибавляет двойку к переданному ей числу и возвращает полученный результат. Внутри этой функции используется переменная *x*, доступ к которой снаружи невозможен. К тому же, эта переменная удаляется из памяти каждый раз (во всяком случае, должна удаляться), когда завершается *add_two()*.

- **Область видимости** *Enclosing*

Суть данной области видимости в том, что внутри функции могут быть вложенные функции и локальные переменные, так вот локальная переменная функции для ее вложенной функции находится в *enclosing* области видимости.

Пример.

```
>>> def add_four(a):
        x = 2
        def add_some():
            print("x = " + str(x))
            return a + x
        return add_some()

>>> add_four(5)
x = 2
7
```

В данном случае переменная *x* имеет область видимости *enclosing* для функции *add_some()*.

- **Область видимости** *Global*

Переменные области видимости *global* – это глобальные переменные уровня модуля (модуль – это файл с расширением *.py*).

Пример.

```
>>> x = 4
>>> def fun():
        print(x+3)

>>> fun()
7
```

В приведенном выше коде переменная *x* – это *global* переменная. Доступ к ней можно получить из любой функции объявленной в данном модуле. Но если мы этот модуль импортируем в каком-то другом модуле, то *x* для него уже не будет переменной уровня *global*.

- **Область видимости** *Built-in*

Уровень *Python* интерпретатора. В рамках этой области видимости находятся функции *open*, *len* и т. п., также туда входят исключения. Эти сущности доступны в любом модуле *Python* и не требуют предварительного импорта. *Built-in* – это максимально широкая область видимости.

Как уже было сказано выше, каждый раз, когда мы вызываем функцию, у нее создаются локальные переменные (если они у нее есть), а после завершения – уничтожаются, при очередном вызове эта процедура повторяется. Можно ли сделать так, чтобы после завершения работы функции, часть локальных переменных не уничтожалась, а сохраняла свои значение до следующего запуска? Да, это можно сделать!

Локальная переменная не будет уничтожена, если на нее где-то останется “живая” ссылка, после завершения работы функции. Эту ссылку может сохранять вложенная функция. Функции построенные по такому принципу могут использоваться для построения специализированных функций, т.е. являются фабриками функций. Далее будет рассмотрен вопрос создания и использования замыканий в Python, которые как раз и использую эту идею.

Как использовать замыкания в Python?

Для начала разберем следующий пример.

```
>>> def mul(a, b):
        return a * b

>>> mul(3, 4)
12
```

Функция *mul()* умножает два числа и возвращает полученный результат. Если мы ходим на базе нее решить задачу: “умножить число на пять”, то в самом простом случае, можно вызывать *mul()*, передавая в качестве первого аргумента пятерку.

```
>>> mul(5, 2)
10

>>> mul(5, 7)
35
```

Это неудобно. На самом деле мы можем создать новую функцию, которая будет вызывать *mul()*, с пятеркой и ещё одним числом, которое она будет получать в качестве своего единственного аргумента.

```
>>> def mul5(a):
        return mul(5, a)

>>> mul5(2)
10

>>> mul5(7)
35
```

Уже лучше, но все равно пока не достаточно гибко, т. к. в следующий раз, когда нужно будет построить умножитель на семь, нам придется создавать новую функцию. Для решения этой проблемы воспользуемся замыканием.

```
>>> def mul(a):
        def helper(b):
            return a * b
        return helper
```

Вычислим выражение “5 * 2 = ?” с помощью этой функции.

```
>>> mul(5)(2)
  10
```

Создадим функцию – аналог *mul5()*.

```
>>> new_mul5 = mul(5)

>>> new_mul5
<function mul.<locals>.helper at 0x000001A7548C1158>

>>> new_mul5(2)
10

>>> new_mul5(7)
35
```

Вызывая *new_mul5(2)*, мы фактически обращаемся к функции *helper()*, которая находится внутри *mul()*. Переменная **a**, является локальной для *mul()*, и имеет область *enclosing* в *helper()*. Несмотря на то, что *mul()* завершила свою работу, переменная **a** не уничтожается, т.к. на нее сохраняется ссылка во внутренней функции, которая была возвращена в качестве результата.

Рассмотрим ещё один пример.

```
>>> def fun1(a):
        x = a * 3
        def fun2(b):
            nonlocal x
            return b + x
        return fun2

>>> test_fun = fun1(4)

>>> test_fun(7)
19
```

В функции *fun1()* объявлена локальная переменная *x*, значение которой определяется аргументом *a*. В функции *fun2()* используются эта же переменная *x*, *nonlocal* указывает на то, что эта переменная не является локальной, следовательно, ее значение будет взято из ближайшей области видимости, в которой существует переменная с таким же именем. В нашем случае – это область *enclosing*, в которой этой переменной *x* присваивается значение *a * 3*. Также как и в предыдущем случае, на переменную *x* после вызова *fun1(4)*, сохраняется ссылка, поэтому она не уничтожается.

Свойство замыкания – средство для построения иерархических данных

Сразу хочу сказать, что “свойство замыкания” – это не то замыкание, которое мы разобрали выше. Начнем разбор данного термина с математической точки зрения, а точнее с алгебраической. Предметом алгебры является изучение алгебраических структур – множеств с определенными на них операциями. Под множеством обычно понимается совокупность определенных объектов. Наиболее простым примером числового множества, является множество натуральных чисел. Оно содержит следующие числа: 1, 2, 3, ... и т.д. до бесконечности. Иногда, к этому множеству относят число ноль, но мы не будем этого делать. Над элементами этого множества можно производить различные операции, например сложение:

$$1 + 2 = 3. \tag{1}$$

Какие бы натуральные числа мы не складывали, всегда будем получать натуральное число. С умножением точно также. Но с вычитанием и делением это условие не выполняется.

$$2 - 5 = -3. \tag{2}$$

Среди натуральных чисел нет числа -3, для того, чтобы можно было использовать вычитание без ограничений, нам необходимо расширить множество натуральных чисел до множества целых чисел:

$$-\infty, \ldots, -2, -1, 0, 1, 2, \ldots, +\infty. \tag{3}$$

Таким образом, можно сказать, что множество натуральных чисел замкнуто относительно операции сложения – какие бы натуральные числа мы не складывали, получим натуральное число, но это множество не замкнуто относительно операции вычитания.

Теперь перейдем с уровня математики на уровень функционального программирования. Вот как определяется “свойство замыкания” в книге “Структура и интерпретация компьютерных программ” Айбельсона Х., Сассмана Д. Д. : “В общем случае, операция комбинирования объектов данных обладает свойством замыкания в том случае, если результаты соединения объектов с помощью этой операции сами могут соединяться этой же операцией”.

Это свойство позволяет строить иерархические структуры данных. Покажем это на примере кортежей в *Python*.

Создадим функцию *tpl()*, которая на вход принимает два аргумента и возвращает кортеж. Эта функция реализует операцию “объединения элементов в кортеж”.

```
>>> tpl = lambda a, b: (a, b)
```

Если мы передадим в качестве аргументов числа, то, получим простой кортеж.

```
>>> a = tpl(1, 2)
>>> a
(1, 2)
```

Эту операцию можно производить не только над числами, но и над сущностями, ей же и порожденными.

```
>>> b = tpl(3, a)
>>> b
(3, (1, 2))

>>> c = tpl(a, b)
>>> c
((1, 2), (3, (1, 2)))
```

Таким образом, в нашем примере кортежи оказались замкнуты относительно операции объединения *tpl*. Вспомните аналогию с натуральными числами, замкнутыми относительно сложения.

Аппаратура и материалы

1. Компьютерный класс общего назначения с конфигурацией ПК не хуже рекомендованной для ОС Windows 10 с подключением к глобальной сети Интернет.

2. Операционная система Windows 10.
3. Система контроля версий Git.
4. Браузер для доступа к web-сервису GitHub, рекомендован к использованию Google Chrome.
5. Дистрибутив языка программирования Python, включающий набор популярных библиотек Anaconda.
6. Интегрированная среда разработки PyCharm Community Edition.

Указания по технике безопасности

При работе на ЭВМ без разрешения руководителя занятия запрещается:

- подавать (снимать) напряжение на ПЭВМ и электрические розетки с распределительного щита;
- включать и выключать блоки питания ПЭВМ и мониторы;
- извлекать ПЭВМ из защитного кожуха;
- устранять неисправности, возникшие в ходе выполнения лабораторной работы.

Методика и порядок выполнения работы

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.
3. Выполните клонирование созданного репозитория.
4. Дополните файл `.gitignore` необходимыми правилами для работы с IDE PyCharm.
5. Организуйте свой репозиторий в соответствие с моделью ветвления git-flow.
6. Создайте проект PyCharm в папке репозитория.
7. Проработать примеры лабораторной работы.
8. Выполнить индивидуальное задание.
9. Зафиксируйте изменения в репозитории.
10. Добавьте отчет по лабораторной работе в *формате PDF* в папку *doc* репозитория. Зафиксируйте изменения.
11. Выполните слияние ветки для разработки с веткой *master*/*main*.
12. Отправьте сделанные изменения на сервер GitHub.
13. Отправьте адрес репозитория GitHub на электронный адрес преподавателя.

Индивидуальное задание

Составить программу с использованием замыканий для решения задачи. Номер варианта определяется по согласованию с преподавателем.

1. Используя замыкания функций, определите вложенную функцию, которая бы увеличивала значение переданного параметра на 3 и возвращала бы вычисленный результат. Вызовите внешнюю функцию для получения ссылки на внутреннюю функцию и присвойте ее переменной с именем cnt. Затем, вызовите внутреннюю функцию через переменную `cnt` со значением `k`, введенным с клавиатуры.

2. Используя замыкания функций, объявите внутреннюю функцию, которая заключает строку `s` (`s` – строка, параметр внутренней функции) в произвольный тег, содержащийся в переменной `tag` – параметре внешней функции. Далее, на вход программы поступает две строки: первая с тегом, вторая с некоторым содержимым. Вторую строку нужно поместить в тег из первой строки с помощью реализованного замыкания. Результат выведите на экран.

3. Используя замыкания функций, объявите внутреннюю функцию, которая преобразует строку из списка целых чисел, записанных через пробел, либо в список, либо в кортеж. Тип коллекции определяется параметром `type` внешней функции. Если `type = 'list'`, то используется список, иначе – кортеж. Далее, на вход программы поступает две строки: первая – это значение для параметра type; вторая – список целых чисел, записанных через пробел. С помощью реализованного замыкания преобразовать эту строку в соответствующую коллекцию. Результат работы замыкания выведите на экран.
4. Используя замыкания функций, объявите внутреннюю функцию,
которая из переданного ей списка строк формирует многострочную строку вида:

```
<ol>
<li>строка_1</li>
…
<li>строка_N</li>
</ol>
```

и возвращает ее. Где строка*1, строка*2, ... - это строки из переданного функции списка. Вызовите внутреннюю функцию замыкания и отобразите на экране результат ее работы.

5. Используя замыкания функций, объявите внутреннюю функцию, которая принимает в качестве параметров фамилию и имя, а затем, заносит в шаблон эти данные. Сам шаблон – это строка, которая передается внешней функции и, например, может иметь такой вид: «Уважаемый %F%, %N%! Вы делаете работу по замыканиям функций.» Здесь %F% - это фрагмент куда нужно подставить фамилию, а %N% - фрагмент, куда нужно подставить имя. (Шаблон может быть и другим, вы это определяете сами). Здесь важно, чтобы внутренняя функция умела подставлять данные в шаблон, формировать новую строку и возвращать результат. Вызовите внутреннюю функцию замыкания и отобразите на экране результат ее работы.
6. Используя замыкания функций, объявите внутреннюю функцию, которая бы все повторяющиеся символы заменяла одним другим указанным символов. Какие повторяющиеся символы искать и на что заменять, определяются параметрами внешней функции. Внутренней функции передается только строка для преобразования. Преобразованная (сформированная) строка должна возвращаться внутренней функцией. Вызовите внутреннюю функцию замыкания и отобразите на экране результат ее работы.
7. Используя замыкания функций, объявите внутреннюю функцию, которая на основе двух параметров вычисляет площадь фигуры. Какой именно фигуры: треугольника или прямоугольника, определяется параметром `type` внешней функции. Если `type` принимает значение 0, то вычисляется площадь треугольника, а иначе – прямоугольника. По умолчанию параметр `type` должен быть равен 0. Вычисленное значение должно возвращаться внутренней функцией. Вызовите внутреннюю функцию замыкания и отобразите на экране результат ее работы.
8. Используя замыкания функций, объявите внутреннюю функцию, которая принимает два параметра `a`, `b`, а затем, возвращает строку в формате: «Для значений a, b функция f(a,b) = <число>» где `число` – это вычисленное значение функции `f`. Ссылка на `f` передается как аргумент внешней функции. Вызовите внутреннюю функцию замыкания и отобразите на экране результат ее работы. Функцию `f` придумайте самостоятельно (она должна что то делать с двумя параметрами `a`, `b` и возвращать результат).

9. Используя замыкания функций, объявите внутреннюю функцию, которая принимает в качестве аргумента коллекцию (список или кортеж) и возвращает или минимальное значение, или максимальное, в зависимости от значения параметра `type` внешней функции. Если `type` равен «max», то возвращается максимальное значение, иначе – минимальное. По умолчанию `type` должно принимать значение «max». Вызовите внутреннюю функцию замыкания и отобразите на экране результат ее работы.
10. Используя замыкания функций, объявите внутреннюю функцию, которая принимает в качестве аргумента список целых чисел и удаляет из него все четные или нечетные значения в зависимости от значения параметра `type`. Если `type` равен «even», то удаляются четные значения, иначе – нечетные. По умолчанию `type` должно принимать значение «even». Вызовите внутреннюю функцию замыкания и отобразите на экране результат ее работы.

Содержание отчета и его форма

Отчет по лабораторной работе оформляется электронно в формате PDF, должен содержать ответы на контрольные вопросы, ссылку на репозиторий с которым выполнялась работа, скриншоты IDE PyCharm, скриншоты результатов работы программ.

Вопросы для защиты работы

1. Что такое замыкание?
2. Как реализованы замыкания в языке программирования Python?
3. Что подразумевает под собой область видимости Local?
4. Что подразумевает под собой область видимости Enclosing?
5. Что подразумевает под собой область видимости Global?
6. Что подразумевает под собой область видимости Build-in?
7. Как использовать замыкания в языке программирования Python?
8. Как замыкания могут быть использованы для построения иерархических данных?