ЛАБОРАТОРНАЯ РАБОТА

Тема: ***Функции как инструмент структурирования программного кода***

Цель работы: Освоить простейшие приемы декомпозиции программ

Отчет:

Содержание

Приложение "Поиск палиндромов"

Разрабатывается простой вариант прикладной программы (англ. `application` — "приложение") "`Поиск палиндромов`".

В будущем задача будет усложнена за счет обработки массива слов и уточнения поведения программы.

Общая (словесная, вербальная) постановка задачи

Разрабатывается приложение для проверки, является ли слово палиндромом.

Пользователю предлагается ввести слово.

Приложение выполняет анализ и сообщает пользователю результат: палиндром / не палиндром.

Формальная (точная) постановка задачи

Приложение предназначено для анализа на принадлежность к палиндромам *одного* слова (символьной строки).

Слово для анализа вводится в диалоге с пользователем *с клавиатуры*.

Приложение выполняет контроль правильности ввода (*в первом варианте* программы это требование *не учитывается*).

Приложение проверяет, является ли слово палиндромом.

На основании результата проверки формируется итоговое *текстовое* сообщение одного из двух видов:

`Слово 'заданное_слово' — палиндром` или

`Слово 'заданное_слово' — не палиндром`.

Сообщение выводится *на экран*.

Проектирование приложения

а) ***Разбиение на функции (декомпозиция).***

В процедурном программировании *декомпозиция* —— это метод замены решения одной большой задачи решением серии *связанных между собой* меньших, но более простых задач.

Разбиение приложения на отдельные функции (функциональные модули) называется *функциональной декомпозицией*.

Замечание

Композиция — означает составление целого из частей. Приставка "де" в слове "декомпозиция" означает действие, противоположное композиции.

Программа "Поиск палиндромов" должна решать несколько отдельных подзадач: *ввод* исходных данных, *вычисления*, *вывод* результатов. Решение каждой из подзадач выполняется отдельной функцией.

`get_word()` — в диалоге с приложением пользователь предлагает слово для анализа.

`is_palindrome()` — выполняет проверку, является ли заданное пользователем слово палиндромом.

`create_message()` — формирует требуемый вид строки с итоговым сообщением.

`check_palindrome()` — главная функция программы, где реализуется логика работы приложения. Функция выводит название программы "`Поиск палиндромов`", а затем обеспечивает выполнение операций в требуемой последовательности: `get_word` — `is_palindrome` — `create_message` — `вывод сообщения`. Функция `check_palindrome()` не имеет аргументов и не содержит операторов возврата.

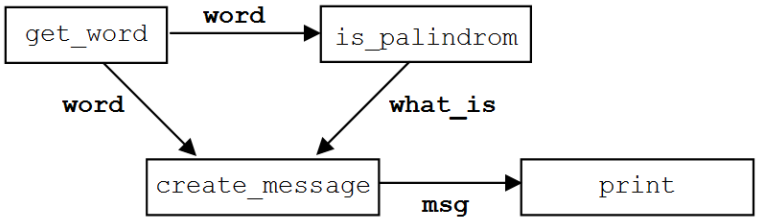
б) *Организация обмена данными между функциями.*

Каждая функция решает отдельную подзадачу. Чтобы объединить их усилия, приложение должно обеспечить передачу информации между функциями.

Слово, введенное пользователем при вызове функции `get_word()` (обозначим переменной `word`) потребуется для выполнения проверки функцией `is_palindrome()` и подготовки строки для вывода результатов функцией `create_message()`.

Результат выполнения проверки слова функцией `is_palindrome()` (обозначим переменной `what_is`) используется функцией `create_message()` для формирования сообщения о результате.

Текст сформированного `create_message()` сообщения (обозначим переменной `msg`) используется в `check_palindrome()` для вывода экран с помощью функции `print`.



Обмен информацией между функциями можно реализовать двумя методами. В первом для этого используются формальные параметры (аргументы) и возвращаемые значения функций, а во втором — переменные, определенные в глобальном пространстве (см. справочную информацию).

ЗАДАНИЕ 1 (*Использование аргументов и возвращаемых значений функций*)

Уточним проектное решение для случая обмена данными с помощью возвращаемых значений функций. Предполагается, что результат своей работы каждая функция возвращает как выражение в операторе возврата `return`.

Так как вся вычислительная логика программы сосредоточена в главной функции `check_palindrome`, то возвращаемые значения будем сохранять в локальных переменных *этой функции*.

`get_word()` — в диалоге с программой пользователь вводит слово для анализа, которое становится *возвращаемым значением функции*. Функция `get_word()` вызывается в функции `check_palindrome`. Полученное значение сохраняется во вспомогательной переменной `word`.

`is_palindrome(word)` — выполняет проверку, является ли переданное ей слово `word` палиндромом или нет. Функция вызывается в функции `check_palindrome` и возвращает сообщение "`палиндром`" или "`не палиндром`", которое запоминается во вспомогательной переменной `what_is`.

`message(word, what_is)` — возвращает строку с описанием результата. Для этого используется оператор форматного вывода `%`. В функции `check_palindrome` возвращенная `message` строка запоминается в переменной `msg`.

`check_palindrome()` — главная функция программы. В начале своей работы эта функция выводит, название программы "`Поиск палиндромов`". Затем в ней последовательно вызываются функции `get_word`, `is_palindrome`, `message` и встроенная функция `print`, с помощью которой на экран выводится итоговое сообщение. Функция не имеет аргументов и не содержит операторов возврата.

Определения функций сами по себе пассивны. Чтобы программа начала выполняться требуется в нее добавить всего одну инструкцию с *вызовом* главной функции `check_palindrome()`.

Замечание

В структуре программы отдельными компонентами представлен

- *пользовательский интерфейс приложения* — функции `get_word()` и `create_message()`, обеспечивающие ввод и подготовку вывода информации, и
- *расчетная часть*, бизнес-логика приложения — функция `is_palindrome()`.

В дальнейшем такое построение программы упростит её модификацию для улучшения разрабатываемого приложения.

Последовательность решения задачи

1. Запустить IDLE, создать и сохранить новый файл исходного кода Питона **`palindrome1.py`**.

2. Определить в программе функцию **`get_word`** для ввода анализируемой строки.

Функция предлагает пользователю подсказку "`Введите слово: `", запоминает введенный пользователем текст во вспомогательной переменной (имя дать по своему усмотрению) и возвращает значение этой переменной.

3. Протестировать работу функции get_word, добавив в программу инструкции с вызовом функции и выводом на экран возвращаемого ею значения. При успешном тестировании закомментировать добавленные инструкции.

4. Определить в программе функцию **`is_palindrome`** с одни аргументом — строкой, содержащей проверяемое слово. Если слово является палиндромом, то функция *возвращает* строку "`палиндром`", иначе — "`не палиндром`". Для построения возвращаемого значения использовать оператор `return` с логическим выражением (см. лаб. 5).

5. Протестировать работу функции `is_palindrome`, добавив в программу инструкции с её вызовами и выводом возвращаемого значения. В одном случае передать функции палиндром, в другом — не палиндром (например, '`доход`' и '`поход`'). При успешном тестировании закомментировать добавленные инструкции.

6. Определить в программе функцию **`create_message()`** с двумя аргументами:

- строкой текста `word`, введенного пользователем, и
- результатом проверки слова на палиндром.

Функция готовит и *возвращает* строку с сообщением о результатах анализа слова. Для этого из статического текста и спецификаций форматов для переменных `word` и `what_is` (какой они имеют тип?) создается форматная строка, которой с помощью оператора `%` передаются значения этих переменных.

"Слово ... — палиндром" или "Слово ... — не палиндром".

7. Протестировать работу функции `create_message()`, добавив в программу инструкции с её вызовами и выводом возвращаемого значения. В контрольном примере можно передать функции две произвольные строки. При успешном тестировании закомментировать добавленные инструкции.

8. В определение главной функции включаются следующие инструкции:

вызов `get_word()` и запоминание введенного слова во вспомогательной переменной `word`,

вызов `is_palindrome()`, при котором функции передается значение переменной `word`, и запоминание результата во вспомогательной переменной `what_is`;

вызов функции `create_message()`, которой передаются значения переменных `word` и `what_is`. и запоминание результата во вспомогательной переменной `msg`.

9. Протестировать работу программы, используя слова "наган" и "таган".

10. Протестировать программу в случае, когда пользователь вводит пустую строку. Какой будет получен результат?

Глобальное и локальные пространства имен

Переменные в Питоне создаются в момент присваивания им первого значения. Если переменная создана вне функций, то она автоматически попадает в глобальное (т.е. общее для всей программы) пространство имен.

Каждая определяемая в программе *функция* формирует свое подпространство имен (локальное пространство), вложенное в глобальное. Это *локальное* пространство создается каждый раз при вызове функции и уничтожается, когда происходит выход из функции.

При создании функции в ней будут доступны переменные, ранее определенные в глобальном пространстве.

Когда *в функции* создается *новая* переменная, то она автоматически помещается в локальное пространство имен функции. Поэтому всякое присваивание значения переменной *в функции* будет рассматриваться, как работа с локальной переменной.

Если в глобальном пространстве имен ранее уже существовала одноименная переменная, то локальная переменная "затеняет" её, и глобальная становится недоступной в функции.

Чтобы можно было менять значения глобальных переменных в теле функции, эти переменные должны быть предварительно объявлены в функции с помощью ключевого слова `global`.

В функции можно создать новую *глобальную* переменную, если сначала объявить её с помощью `global`, а потом присвоить значение (т.е. создать переменную).

Замечания

1. В одной инструкции *нельзя* совмещать объявление `global` с присваиванием значения переменной (т.е с её созданием).

2. В одном объявлении `global` может быть указано через запятую несколько имен.

3. Объявления `global` не связаны с выделением памяти, они уточняют пространство имен, в которое будет помещена переменная.

ЗАДАНИЕ 2 (*Глобальное и локальные пространства имен*)

1. Создать файл `namespaces.py`.

2. На глобальном уровне определить переменную `var1` со значением "глобальная".

3. Определить функцию `my_func` без аргументов и телом, состоящим из одной инструкции `print(var1)`.

4. Добавить в программу инструкции с вызовом `my_func` и еще одним выводом (вне функции) значения переменной `var1`. Что получено и почему?

5. В начало функции `my_func` добавить присваивание переменной `var1` значения "`локальная`". Вновь выполнить программу. Что получено и почему?

6. В начало функции `my_func` добавить объявление переменной `var1` глобальной, т.е. `global var1`. Вновь выполнить программу. Что получено и почему?

7. В функции `my_func` присвоить значение еще одной переменной: `var2="вторая"`.

В *конец программы* (вне функции) добавить инструкцию `print(var2)`. Выполнить программу. Что получено и почему?

8. В начало функции `my_func` добавить объявление `global var2`. Выполнить программу. Что изменилось и почему?

ЗАДАНИЕ 3 (*Поиск палиндромов — использование глобальных переменных*)

Для решения задачи используем обмен данными с помощью глобальных переменных. Идея состоит в том, что переменные `word` и `what_is` перед созданием объявляются в `get_word()` и `is_palindrome()` как глобальные. Тогда они сразу становятся доступными в остальных функциях программы. После запоминания результатов выполнения функций в этих переменных, нам больше не нужно передавать результаты с помощью `return` и передавать их далее через аргументы.

Замечание

Почему в задании не предложено запоминать в глобальной переменной результат работы функции `create_message`? Это вопрос конструирования программы, который может быть решен по-разному. В данном случае по смыслу программы переменные `word` и `what_is` имеют самостоятельное значение, они несут существенную информацию и, вообще говоря, могут использоваться много раз. А строка для вывода на экран — вспомогательная и предназначенная для разового применения. Поэтому можно обойтись без её запоминания в промежуточной переменной.

1. Создать копию файла `palindrome1.py` под именем **`palindrome2.py`**.

Чтобы реализовать новую схему обмена информацией, нужно внести следующие изменения.

а) В определениях функций `get_word()`, `is_palindrome()` и `create_message()` удалить формальные параметры.

б) В определениях функций `get_word()` и `is_palindrome()` переменные `word` и `what_is` объявляются глобальными.

в) В функциях `get_word()` и `is_palindrome()` удалить операторы возврата.

г) В функции теперь `check_palindrome()` нет необходимости запоминать значения, возвращавшиеся раньше функциями `get_word()` и `is_palindrome()`. Удалить присваивание значений переменным.

2. Протестировать программу, используя слова "`наган`" и "`таган`".

Простейшая форма условного оператора

В простейшем случае условный оператор строится из *строки с условием* и *тела*.

Строка с условием содержит

- ключевого слова `if`,
- следующего за ним выражения, результат которого интерпретируется как `True` или `False`
- двоеточия.

Тело — это блок программного кода, который вводится с отступом и который будет выполнен, если результатом выражения будет интерпретироваться как `True`.

```
if логическое_выражение:
        тело
        оператора
инструкция после условного оператора
```

Замечание

В полной форме инструкция `if` для выполнения дополнительных проверок предусматривает наличие одно или более предложение `elif` (т.е. `else if`) и предложения `else`, описывающего действия, когда не выполнено ни одно из условий

Добавление контроля ввода слов

Далее основным вариантом программы считается `palindrome1.py`. Для создания улучшенного варианта приложения используется копия этого файла под именем `palindrome3.py`.

При тестировании приложения было выявлено, что оно допускает ввод пустой строки, что не имеет практического смысла и, скорее всего, является свидетельством ошибки пользователя. Поэтому приложение должно отслеживать такую ситуацию и выдавать пользователю соответствующее сообщение.

Простейший вариант поведения приложения: при вводе пустой строки пользователь получает сообщение "`Ошибка ввода: пустая строка!!!`" и программа заканчивает свою работу.

ЗАДАНИЕ 4 (*Контроль за правильностью ввода*)

Так как речь идет о контроле ошибки при вводе, то изменения в программе коснутся только функции `get_word()`.

Используя короткую форму условного оператора реализовать в ней следующий алгоритм: если введена пустая строка (какими способами можно в этом убедиться?), то вывести сообщение об ошибке и завершить выполнение программы.

Для завершения работы программы используется инструкция `sys.exit()`, для которой нужно предварительно импортировать стандартный модуль Питона `sys`. С этой целью в начало программы нужно поместить инструкцию

```
import sys
```

Вопросы для самоконтроля

1. В чем основной смысл определения функции? Из каких частей состоит определение функции?
2. Какую структуру имеет определение функции?
3. Что понимается под операторными скобками и что является их альтернативой в Питоне?
4. Какие действия и в каком порядке выполняются вызовы функций?
5. Когда завершается выполнение функции?
6. Что и в каком случае возвращает функция?
7. Что понимается под декомпозицией и функциональной декомпозицией?
8. Чем отличаются глобальное и локальное пространства имен?
9. Можно ли переменную, определенную в функции, поместить в глобальное пространство имен?
10. Что происходит, если переменной, получившей значение на глобальном уровне, присваивается значение в функции?

11. Какой синтаксис у простейшей формы условного оператора?

Справочная информация

Определения функций

Определение функции — это присваивание имени группе инструкций программы. Выполнение этих инструкций происходит при *вызовах* функции по имени. Вызовы функций можно выполнять только после того, как функция определена.

Определение функции состоит из заголовка и тела.

Заголовок состоит

- из служебного слова `def` (от англ. `definition` — определение),
- имени функции, выбранного по обычным правилам для идентификаторов,
- пары круглых скобок с пустым или непустым списком *формальных параметров* (имён аргументов), перечисленных через запятую, и
- двоеточия.

Тело функции — это блок инструкций, который должен быть выделен. В Питоне нет операторных скобок, таких как служебные слова `begin` и `end` в Паскале или фигурные скобки в Си.

Для выделения тела функции каждая из его логические строк должна иметь один и тот же отступ по отношению к позиции строки заголовка (рекомендуется использовать для отступа на новый уровень 4 символа).

```
def имя_функции(список аргументов):
    тело
    функции
инструкция после определения функцией
```

Вызов функции имеет вид

```
имя_функции(список выражений)
```

В круглых скобках через запятую перечисляются выражения, значения которых называют *фактическими параметрами* вызова функции.

Замечание

Формальные параметры — это локальные (т.е. собственные) переменные функции, которые хранят значения фактических параметров, которые передаются при вызове функции и используются при выполнении инструкций тела функции.

Порядок выполнения вызовов функций:

- вычисляются значения фактических параметров,
- эти значения присваиваются соответствующим формальным параметрам,
- выполняются инструкции, помещенные в тело функции,
- вычисляется возвращаемое значение.

Работа функции завершается, когда

- исполнена последняя инструкция тела функции или
- встретилась одна из инструкций *оператора возврата*: `return выражение` или просто `return`.

Вызов функции всегда дает некоторый результат, который называется *возвращаемым значением* функции.

Содержательное возвращаемое значение появляется при выполнении оператора возврата `return` с выражением. В этом случае вычисляется и возвращается значение заданного выражения.

В остальных случаях (встретился оператор `return` без выражения или были исчерпаны все инструкции в теле) функция возвращает фиктивное значение `None`.

Использования объявлений global

Глобальные имена – это имена, которые определены на верхнем уровне основного модуля.

Имена, определяемые внутри инструкции `def` (*локальные*), видны только программному коду внутри инструкции `def`. К этим именам нельзя обратиться за пределами функции.

При присваивании в функции значений новым переменным, не объявленные глобальными, эти переменные помещаются в локальное пространство имен. Локальное пространство имен *функции* создается при ее вызове и удаляется, когда функция возвращает управление в основную часть программы.

Глобальная область имен охватывает все локальные подпространства, и поэтому глобальные имена *видны* во всех функциях. Поэтому в теле функций можно обращаться к глобальным переменным без объявления их глобальными.

Имена глобальных переменных должны объявляться в функции с помощью `global`, *только*, если им внутри функции будут присваиваться значения. Причем таким способом может быть создана даже новая глобальная переменная.

Если переменная с одним и тем же именем определена как глобальная, так и локальная, то в каждой области будет использоваться своя переменная.