МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания

по выполнению практических работ по дисциплине «Основы лингвистического программирования»

Направление подготовки	45.04.02 Лингвистика
Направленность (профиль)	Современные методы прикладной лингвистики и перевода
Год начала обучения	2026
Форма обучения	очная
Реализуется в семестре	2

Ставрополь
2026

Введение

Методические рекомендации по выполнению практических работ по дисциплине «Основы лингвистического программирования» разработаны в соответствии с рабочей программой дисциплины по направлению 45.04.02 Лингвистика, профиль – Современные методы прикладной лингвистики и перевода.

Практические задания разработаны в соответствии с рабочей программой дисциплины «Основы лингвистического программирования», целью которой является систематическое изложение основ программирования лингвистических алгоритмов с применением языка программирования Python, а также развитие соответствующих практических навыков.

Целью практических занятий является закрепление теоретических знаний и приобретение практических умений и навыков, необходимых для освоения базовых принципов программирования алгоритмов обработки текстов на естественных языках.

Методические рекомендации по каждой практической работе имеют теоретическую часть, необходимую для выполнения практических заданий. Практические задания органично сочетаются с теоретическими знаниями.

Практическое занятие 1.

Тема: Теоретические основания лингвистического программирования. Введение в язык программирования Python.

Цель: освоение студентами базовых принципов программирования алгоритмов обработки текстов на естественных языках с применением языка программирования Python.

Реализуемые компетенции:

ПК-4 - Способен применять лингводидактические знания при решении исследовательских, педагогических и прикладных задач, комплексно работать с лингвистической информацией в педагогической деятельности.

Актуальность: актуальность изучения темы определяется практической целесообразностью использования лингвистических алгоритмов для оптимизации деятельности лингвиста-практика в части обработки текстовой информации на естественных языках.

Вопросы и задания для собеседования (по материалам: URL: http://pep8.ru/doc/tutorial-3.1/).

Выполните следующие задания.

Интерпретатор Python после установки располагается, обычно, по пути /usr/local/bin/python3.1 — на тех компьютерах, где этот путь доступен. Добавление каталога /usr/local/bin к пути поиска Unix-шелла (переменная PATH) позволит запустить интерпретатор набором команды

python3.1

прямо из шелла. Поскольку выбор каталога, в котором будет обитать интерпретатор, осуществляется при его установке, то возможны и другие варианты — посоветуйтесь с вашим Python-гуру или системным администратором. (Например, путь /usr/local/python тоже популярен в качестве альтернативного расположения.)

На машинах с ОС Windows, инсталляция Python обычно осуществляется в каталог C:\Python31, но и он может быть изменён во время установки. Чтобы добавить этот каталог к вашему пути поиска, вы можете набрать в окне DOS следующую команду, в ответ на приглашение:

set path=%path%;C:\python31

При наборе символа конца файла (Ctrl-D в Unix, Ctrl-Z в Windows) в ответ на основное приглашение интерпретатора, последний будет вынужден закончить работу с нулевым статусом выхода. Если это не сработает — вы можете выйти из интерпретатора путём ввода следующих команд:

import sys; sys.exit()

Особенности редактирования строк в интерпретаторе не оказываются обычно чересчур сложными. Те, кто установил интерпретатор на машину Unix, потенциально имеют поддержку библиотеки GNU readline, обеспечивающей усовершенствованное интерактивное редактирование и сохранение истории. Самый быстрый, наверное, способ проверить, поддерживается ли расширенное редактирование командной строки, заключается в нажатии Ctrl-P в ответ на первое полученное приглашение Python. Если вы услышите сигнал — значит вам доступно редактирование командной строки — тогда обратитесь к Приложению об Интерактивном редактировании входных данных за описанием клавиш. Если на ваш взгляд ничего не произошло или отобразился символ ^P — редактирование командной строки недоступно — удалять символы из текущей строки возможно будет лишь использованием клавиши Backspace.

Интерпретатор ведёт себя сходно шеллу Unix: если он вызван, когда *стандартный ввод* привязан к устройству tty — он считывает и выполняет команды в режиме диалога; будучи вызванным с именем файла в качестве параметра или с файлом, назначенным на *стандартный ввод* — он читает и выполняет сценарий из этого файла.

Другой способ запустить интерпретатор — директива **python -c** *команда* [arg] ... — при её использовании поочередно выполняются инструкции(-ция) из *команды* (как при использовании опции **-c** Unix-шелла). В связи с тем, что инструкции Python часто содержат пробелы или другие специальные для шелла символы, рекомендуется заключать *команды* полностью в одинарные кавычки.

Некоторые модули Python оказываются полезными при использовании их в качестве сценариев. Они могут быть запущены в этом виде командой **python -m** *модуль* [arg] ... — таким образом исполняется исходный файл модуля (как произошло бы, если бы вы ввели его полное имя в командной строке).

Обратите внимание на различие между командами python file и python <file. В последнем случае запросы на ввод из программы, такие как вызов sys.stdin.read(), удовлетворяются из самого файла. Поскольку файл уже был прочитан до конца ещё до начала выполнения программы — символ конца файла будет обнаружен программой незамедлительно. В большинстве случаев (это, чаще всего, и есть те ситуации, которых вы ожидали) вызовы удовлетворяются независимо от того, файл или устройство привязаны к стандартному вводу интерпретатора Python.

При использовании файла сценария иногда полезно иметь возможность запустить сценарий и затем войти в интерактивный режим. Это может быть сделано через указание параметра **-i** перед именем сценария. (Этот способ не сработает, если сценарий считывается со стандартного ввода — по той самой причине, которая описана в предыдущем абзаце).

Передача параметров

В случае, если интерпретатору известны имя сценария и дополнительные параметры, с которыми он вызван, все они передаются сценарию в переменной sys.argv, представляющей собой список (list) строк. Длина (length) списка — минимум, единица; если не переданы ни имя сценария, ни аргументы — то sys.argv[0] содержит пустую строку. Когда в качестве имени сценария передан '-' (означает *стандартный ввод*), sys.argv[0] устанавливается в '-'. Если используется директива **-c** *команда* — sys.argv[0] содержит '-c'. В случае, если используется директива **-m** *модуль* — то sys.argv[0] устанавливается равным полному имени модуля по расположению. Опции, обнаруженные после сочетаний **-c** *команда* или **-m** *модуль* не обрабатываются интерпретатором Python, но остаются в переменной sys.argv, дабы обеспечить возможность отслеживания в самой команде или в модуле.

Интерактивный режим

Если команды считываются с tty — говорят, что интерпретатор находится в *интерактивном режиме* (режиме диалога). В этом режиме он приглашает к вводу следующей команды, отобразив *основное приглашение* (обычно это три знака «больше-чем» — >>>); в то же время, для *продолжающих строк* выводится *вспомогательное приглашение* (по умолчанию — три точки — ...). Перед выводом первого приглашения интерпретатор отображает приветственное сообщение, содержащее номер его версии и пометку о правах копирования:

```
$ python3.1
Python 3.1a1 (py3k, Sep 12 2007, 12:21:02)
```

```
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Продолжающие строки используются в случаях, когда необходимо ввести многострочную конструкцию. Взгляните, например, на следующий оператор if:

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
... print("Be careful not to fall off!")
...
Be careful not to fall off!
```

Интерпретатор и его окружение

Обработка ошибок

В случае появления ошибки интерпретатор выводит сообщение об ошибке, завершая его стеком вызовов. В интерактивном режиме он снова возвращается в состояние приглашения для ввода команд; если ввод происходит из файла — интерпретатор выходит с ненулевым статусом, сразу после распечатки стека вызовов. (Исключения, обрабатываемые в блоке except оператора try в этом контексте не считаются ошибками.) Некоторые ошибки исключительно фатальны и вызывают собой принудительное завершение работы с ненулевым статусом — это применимо к внутренним противоречиям языка и к некоторым случаям нехватки памяти. Все сообщения об ошибках выводятся в *стандартный поток ошибок* (error stream). Обычный вывод исполняемых команд направляется в *стандартный вывод*.

Нажатие клавиш прерывания процесса (обычно Ctrl-C или DEL), в ответ на приглашение в основном или вспомогательном режиме, отменяет ввод и возвращает вас к основному приглашению. Символ прерывания, набранный во время выполнения какой-либо команды порождает исключение KeyboardInterrupt, которое, в свою очередь, может быть перехвачено оператором try.

Исполняемые сценарии на Python

На Unix-системах семейства BSD сценарии на Python могут быть сделаны исполняемыми, также как и шелл-сценарии, путём добавления следующей строки

```
#! /usr/bin/env python3.1
```

(предполагается, что интерпретатор может быть найден по одному из путей, указанных в пользовательской переменной PATH) в начало сценария и установки файла в исполняемый режим. Символы #! должны быть первыми символами в файле. На некоторых платформах строка должна оканчиваться символом конца строки в стиле Unix ('\n'), а не в стиле Windows ('\r\n'). Заметьте, что символ решётки '#' используется в Python для указания начала комментария.

Исполняемый режим (или разрешение на исполнение) может быть установлен сценарию использованием команды **chmod**:

```
$ chmod +x myscript.py
```

У систем с операционной системой Windows нет такого понятия, как исполняемый режим. Установщик Python автоматически связывает файлы .py с файлом python.exe, таким образом двойной клик на файле Python запустит его в виде сценария. Расширение может быть и .pyw в случае, если окно консоли (которое, обычно, отображается) при запуске сценария подавляется.

Кодировка исходных файлов

По умолчанию, исходники Python считаются созданными в кодировке UTF-8. В этой кодировке в строковых литералах, идентификаторах и комментариях могут быть использованы символы большинства языков мира — учитывая то, что стандартная библиотека Python использует символы ASCII для именования идентификаторов — и этому соглашению должен следовать любой переносимый код. Для корректного отображения всех этих символов, ваш редактор должен опознавать файл как закодированный в UTF-8 и должен использовать шрифт, который содержит все символы, используемые в файле.

Также, есть возможность указать другую кодировку для исходных файлов. Для этого нужно добавить специальный комментарий следом за строкой #!, дабы описать кодировку исходного файла:

```
# -*- coding: encoding -*-
```

Если используется это описание — всё, что находится в этом файле будет опознаваться как имеющее соответствующую кодировку *encoding*, а не UTF-8. Список возможных кодировок представлен в Справочнике по библиотеке — в разделе, описывающем модуль codecs.

Например, если выбранный вами редактор не поддерживает файлы, закодированные UTF-8 и требует применения какой-либо другой кодировки, допустим Windows-1252, вы можете написать:

```
# -*- coding: cp-1252 -*-
```

И, с этого момента, использовать в исходных файлах только символы из таблицы символов Windows-1252. Устанавливающий (отличную от установленной по умолчанию) кодировку специальный комментарий должен являться первой или второй строкой файла.

Интерактивный файл запуска

Если вы используете Python интерактивно — часто бывает удобным выполнить некоторые стандартные команды перед запуском интерпретатора. Вы можете сделать это, установив переменную окружения с именем PYTHONSTARTUP равной имени файла, содержащего ваши команды запуска. Способ схож с использованием файла .profile в Unix-шелле.

Этот файл читается только в интерактивных сессиях, но не в случае считывания команд из сценария, и не в случае, если /dev/tty назначен как независимый источник команд (который в других случаях ведет себя сходно интерактивным сессиям). Файл исполняется в том же пространстве имён, что и исполняемые команды — поэтому объекты и импортированные модули, которые он определяет могут свободно использоваться в интерактивной сессии. Также в этом файле вы можете изменить приглашения: sys.ps1 и sys.ps2.

Если вы хотите прочитать дополнительный файл запуска из текущего каталога — вы можете использовать код вроде:

```
if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())
```

Если вы хотите использовать файл запуска в сценарии — вам нужно будет указать это явно:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
exec(open(filename).read())
```

Список рекомендуемой литературы

Основная литература

1. Саммерфилд М. Python на практике. М.: ДМП Пресс, 2014. - 338 с.

Дополнительная литература

1. Роббинс, А. Изучаем редакторы vi и Vim / А. Роббинс, Э. Хана, Л. Лэмб. – М.: Символ-Плюс, 2013. – 512 с.

2. Чан У. Дж. Python. Создание приложений. - М.: Вильямс, 2013. - 816 с.

3. Horstmann, C., Cornell, G. Core Java Volume I – Fundamentals (9th Edition) / C. Horstmann, G. Cornell. – Prentice Hall, 2012. – 1008 p.

4. Friedl, J.E.F. Mastering Regular Expressions / J.E.F. Friedl. – 3rd. ed. O'Reilly Media, 2012. – 544 p.

Методическая литература

1. Методические рекомендации к самостоятельной работе студентов по дисциплине «Основы программирования лингвистических алгоритмов». Каменский М.В. – Ставрополь, 2026

Интернет-ресурсы

1. Annotation Graph Toolkit (AGTK) – http://agtk.sourceforge.net
2. Applied Linguistics – http://www.appliedlinguistics.org
3. Center for Applied Linguistics – http://www.cal.org
4. Code::Blocks C/C++ IDE – http://www.codeblocks.org
5. Emacs – http://www.gnu.org/software/emacs
6. GATE (General Architecture for Text Engineering) – http://gate.ac.uk
7. GNU Compiler Collection – http://gcc.gnu.org
8. Helsinki Finite-State Technology – http://sourceforge.net/projects/hfst
9. Leopar Language Parser – http://leopar.loria.fr
10. NLTK (Natural Language Toolkit) – http://www.nltk.org
11. Open Natural Language Processing (OpenNLP) – http://opennlp.sourceforge.net
12. Praat – http://www.fon.hum.uva.nl/praat
13. Python Programming Language – http://www.python.org
14. Sed – http://www.gnu.org/s/sed
15. Sonic Visualizer – http://www.sonicvisualiser.org
16. Vi Improved - http://www.vim.org

Практическое занятие 2.

Тема: Синтаксис языка программирования Python. Строки: операции над строками.

Цель: освоение студентами базовых принципов программирования алгоритмов обработки текстов на естественных языках с применением языка программирования Python.

Реализуемые компетенции:

ПК-4 - Способен применять лингводидактические знания при решении исследовательских, педагогических и прикладных задач, комплексно работать с лингвистической информацией в педагогической деятельности.

Актуальность: актуальность изучения темы определяется практической целесообразностью использования лингвистических алгоритмов для оптимизации деятельности лингвиста-практика в части обработки текстовой информации на естественных языках.

Вопросы и задания для собеседования (по материалам: URL: http://pep8.ru/doc/tutorial-3.1/).

Выполните следующие действия и задания.

В приведенных далее примерах, ввод и вывод различаются присутствием и отсутствием приглашений соответственно (приглашениями являются >>> и ...): чтобы воспроизвести пример — вам нужно ввести всё, что следует за приглашением, после его появления; строки, не начинающиеся с приглашений являются выводом интерпретатора. Обратите внимание, что строка, в которой содержится лишь вспомогательное приглашение («... ») означает, что вам нужно ввести пустую строку — этот способ используется для завершения многострочных команд.

Большинство примеров в этом руководстве — даже те, которые вводятся в интерактивном режиме — содержат комментарии. Комментарии в Python начинаются с символа решетки # (hash) — и продолжаются до физического конца строки. Комментарии могут находиться как в начале строки, так и следовать за пробельными символами или кодом — но не содержаться внутри строки. Символ решётки в строке остаётся лишь символом решётки. Поскольку комментарии предназначены для того, чтобы сделать код более понятным, и не интерпретируются Python — при вводе примеров они могут быть опущены.

Несколько примеров:

```
# это первый комментарий
SPAM = 1 # а это второй комментарий
# ... и наконец третий!
STRING = "# Это не комментарий."
```

Использование Python в качестве калькулятора

Давайте опробуем несколько простых команд Python. Запустите интерпретатор и дождитесь появления основного приглашения — >>>. (Это не должно занять много времени.)

Числа

Поведение интерпретатора сходно поведению калькулятора: вы вводите выражение, а в ответ он выводит значение. Синтаксис выражений привычен: операции +, -, * и / работают также как и в большинстве других языков (например, Паскале или C); для группировки можно использовать скобки. Например:

```
>>> 2+2
```

```
4
>>> # Это комментарий
... 2+2
4
>>> 2+2 # а вот комментарий на одной строке с кодом
4
>>> (50-5*6)/4
5.0
>>> 8/5 # При делении целых чисел дробные части не теряются
1.6000000000000001
```

Заметьте: Вы можете получить результат, несколько отличный от представленного: результаты деления с плавающей запятой могут различаться на разных системах. Позже мы расскажем о том, как контролировать вывод чисел с плавающей запятой. Здесь использован наиболее информативный вариант вывода этого значения, а не более легко-читаемый, какой возможен:

```
>>> print(8/5)
1.6
```

Чтобы учебник читался легче, мы будем показывать упрощённый вывод чисел с плавающей точкой и объясним позже, почему эти два способа отображения чисел с плавающей точкой стали различными. Обратитесь к приложению Арифметика с плавающей точкой: Проблемы и ограничения, чтобы ознакомится с подробным обсуждением.

Для получения целого результата при делении целых чисел, отсекая дробные результаты, предназначена другая операция: //:

```
>>> # Деление целых чисел возвращает округлённое к минимальному значение:
... 7//3
2
>>> 7//-3
-3
```

Знак равенства ('=') используется для присваивания переменной какого-либо значения. После этого действия в интерактивном режиме ничего не выводится:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Значение может быть присвоено нескольким переменным одновременно:

```
>>> x = y = z = 0 # Нулевые x, y и z
>>> x
0
>>> y
0
>>> z
0
```

Переменные должны быть *определены* (defined) (должны иметь присвоенное значение) перед использованием, иначе будет сгенерирована ошибка:

```
>>> # попытка получить доступ к неопределённой переменной
... n
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Присутствует полная поддержка операций с плавающей точкой; операции над операндами смешанного типа конвертируют целочисленный операнд в число с плавающей запятой:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Поддерживаются и комплексные числа, добавлением к мнимым частям суффикса j или J. Комплексные числа с ненулевым вещественным компонентом записываются в виде (<*вещественная_часть*>+<*мнимая_часть*>j), или могут быть созданы с помощью функции complex(<*вещественная_часть*>, <*мнимая_часть*>).

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0, 1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Комплексные числа всегда представлены в виде двух чисел с плавающей запятой — вещественной и мнимой частями. Для получения этих частей из комплексного числа z используется z.real и z.imag соответственно.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Функции конвертации (приведения) к вещественным и целым числам (float(), int()) не работают с комплексными числами, так как нет единственно правильного способа сконвертировать комплексное число в вещественное. Используйте функцию abs(z) чтобы получить *модуль* числа (в виде числа с плавающей точкой) или z.real чтобы получить его вещественную часть:

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in ?
TypeError: can`t convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a) # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

В интерактивном режиме последнее выведенное выражение сохраняется в переменной _. Это значит, что если вы используете Python в качестве настольного калькулятора — всегда есть способ продолжить вычисления с меньшими усилиями, например:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Эта переменная для пользователя должна иметь статус *только для чтения*. Не навязывайте ей значение собственноручно — вы создадите независимую переменную с таким же именем, скрывающую встроенную переменную вместе с её свойствами.

Список рекомендуемой литературы

Основная литература

1. Саммерфилд М. Python на практике. М.: ДМП Пресс, 2014. - 338 с.

Дополительная литература

1. Роббинс, А. Изучаем редакторы vi и Vim / А. Роббинс, Э. Хана, Л. Лэмб. – М.: Символ-Плюс, 2013. – 512 с.

2. Чан У. Дж. Python. Создание приложений. - М.: Вильямс, 2013. - 816 с.

3. Horstmann, C., Cornell, G. Core Java Volume I – Fundamentals (9th Edition) / C. Horstmann, G. Cornell. – Prentice Hall, 2012. – 1008 p.

4. Friedl, J.E.F. Mastering Regular Expressions / J.E.F. Friedl. – 3rd. ed. O'Reilly Media, 2012. – 544 p.

Методическая литература

1. Методические рекомендации к самостоятельной работе студентов по дисциплине «Основы программирования лингвистических алгоритмов». Каменский М.В. – Ставрополь, 2026

Интернет-ресурсы

1. Annotation Graph Toolkit (AGTK) – http://agtk.sourceforge.net
2. Applied Linguistics – http://www.appliedlinguistics.org
3. Center for Applied Linguistics – http://www.cal.org
4. Code::Blocks C/C++ IDE – http://www.codeblocks.org

5. Emacs – http://www.gnu.org/software/emacs
6. GATE (General Architecture for Text Engineering) – http://gate.ac.uk
7. GNU Compiler Collection – http://gcc.gnu.org
8. Helsinki Finite-State Technology – http://sourceforge.net/projects/hfst
9. Leopar Language Parser – http://leopar.loria.fr
10. NLTK (Natural Language Toolkit) – http://www.nltk.org
11. Open Natural Language Processing (OpenNLP) – http://opennlp.sourceforge.net
12. Praat – http://www.fon.hum.uva.nl/praat
13. Python Programming Language – http://www.python.org
14. Sed – http://www.gnu.org/s/sed
15. Sonic Visualizer – http://www.sonicvisualiser.org
16. **Vi Improved - http://www.vim.org**

Практическое занятие 3.
Тема: Строки в Python

Цель: освоение студентами базовых принципов программирования алгоритмов обработки текстов на естественных языках с применением языка программирования Python.

Реализуемые компетенции:

ПК-4 - Способен применять лингводидактические знания при решении исследовательских, педагогических и прикладных задач, комплексно работать с лингвистической информацией в педагогической деятельности.

Актуальность: актуальность изучения темы определяется практической целесообразностью использования лингвистических алгоритмов для оптимизации деятельности лингвиста-практика в части обработки текстовой информации на естественных языках.

Вопросы и задания для собеседования (по материалам: URL: http://pep8.ru/doc/tutorial-3.1/).

Выполните следующие действия и задания.

Помимо чисел, Python может работать со строками, которые, в свою очередь, могут быть описаны различными способами. Строки могут быть заключены в одинарные или двойные кавычки:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Интерпретатор выводит результаты операций над строками тем же способом, каким они были введены: обрамляя в кавычки, и, кроме того, экранируя обратными слэшами внутренние кавычки и другие забавные символы — для того, чтобы отобразить точное значение. Строка заключается в двойные кавычки, если строка содержит одинарные кавычки и ни одной двойной, иначе она заключается в одинарные кавычки. Повторимся, функция print() предоставляет более читаемый вывод.

Строковые литералы могут быть разнесены на несколько строк различными способами. Могут быть использованы *продолжающие строки*, с обратным слэшем в качестве последнего символа строки, сообщающим о том, что следующая строка есть продолжение текущей:

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
Note that whitespace at the beginning of the line is\
significant."

print(hello)
```

Обратите внимание, что новые строки все ещё нужно подключать в строку через \n; новая строка, за которой следуют обратный слэш — не обрабатывается. Запуск примера выведет следующее:

```
This is a rather long string containing
several lines of text just as you would do in C.
Note that whitespace at the beginning of the line is significant.
```

Если мы объявим строковой литерал *сырым* (raw) — символы \n не будут конвертированы в новые строки, но и обратный слэш в конце строки, и символ новой строки в исходном коде — будут добавлены в строку в виде данных. Следовательно, код из примера:

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."

print(hello)
```

выведет:

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

Или, строки могут быть обрамлены совпадающей парой тройных кавычек: """ или '''. Окончания строк не нужно завершать тройными кавычками — при этом будут включены в строку.

```
print("""
Usage: thingy [OPTIONS]
-h Display this usage message
-H hostname Hostname to connect to
""")
```

выводит в результате следующее:

```
Usage: thingy [OPTIONS]
-h Display this usage message
-H hostname Hostname to connect to
```

Строки могут конкатенироваться (склеиваться вместе) операцией + и быть повторенными операцией *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Два строковых литерала, расположенные друг за другом, автоматически конкатенируются; первая строка в предыдущем примере также могла быть записана как word = 'Help' 'A'; это работает только с двумя литералами — не с произвольными выражениями, содержащими строки.

```
>>> 'str' 'ing' # <- Так — верно
'string'
>>> 'str'.strip() + 'ing' # <- Так — верно
'string'
>>> 'str'.strip() 'ing' # <- Так — не верно
File "<stdin>", line 1, in ?
'str'.strip() 'ing'
^
SyntaxError: invalid syntax
```

Строки могут быть проиндексированы; также как и в C, первый символ строки имеет индекс 0. Отсутствует отдельный тип для символов; символ является строкой с единичной длиной. Как и в языке программирования Icon, подстроки могут определены через нотацию срезов (slice): два индекса, разделенных двоеточием.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Индексы срезов имеют полезные значения по умолчанию; опущенный первый индекс заменяется нулём, опущенный второй индекс подменяется размером срезаемой строки.

```
>>> word[:2] # Первые два символа
'He'
>>> word[2:] # Всё, исключая первые два символа
'lpA'
```

В отличие от строк в C, строки Python не могут быть изменены. Присваивание по позиции индекса строки вызывает ошибку:

```
>>> word[0] = 'x'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: 'str' object doesn't support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: 'str' object doesn't support slice assignment
```

Тем не менее, создание новой строки со смешанным содержимым — процесс легкий и очевидный:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Полезный инвариант операции среза: s[:i] + s[i:] эквивалентно s.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Вырождения индексов срезов обрабатываются элегантно: чересчур большой индекс заменяется на размер строки, а верхняя граница меньшая нижней возвращает пустую строку.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Индексы могут быть отрицательными числами, обозначая при этом отсчет справа налево. Например:

```
>>> word[-1] # Последний символ
'A'
>>> word[-2] # Предпоследний символ
'p'
>>> word[-2:] # Последние два символа
'pA'
>>> word[:-2] # Всё, кроме последних двух символов
'Hel'
```

Но обратите внимание, что -0 действительно эквивалентен 0 — это не отсчет справа.

```
>>> word[-0] # (поскольку -0 равен 0)
'H'
```

Отрицательные индексы вне диапазона обрезаются, но не советуем делать это с одно-элементными индексами (*не-срезами*):

```
>>> word[-100:]
'HelpA'
>>> word[-10] # ошибка
Traceback (most recent call last):
File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Один из способов запомнить, как работают срезы — думать о них, как об указателях на места между символами, где левый край первого символа установлен в ноль, а правый край последнего символа строки из n символов имеет индекс n, например:

```
+---+---+---+---+---+
| H | e | l | p | A |
+---+---+---+---+---+
0 1 2 3 4 5
-5 -4 -3 -2 -1
```

Первый ряд чисел дает позицию индексов строки от 0 до 5; второй ряд описывает соответствующие отрицательные индексы. Срез от i до j состоит из всех символов между правым и левым краями, отмеченными, соответственно, через i и j.

Для всех индексов, больших или равных нулю, длина среза — разность между индексами, при условии что оба индекса находятся в диапазоне. Например, длина word[1:3] — 2.

Встроенная функция len() возвращает длину строки:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Список рекомендуемой литературы

Основная литература

1. Саммерфилд М. Python на практике. М.: ДМП Пресс, 2014. - 338 с.

Дополительная литература

1. Роббинс, А. Изучаем редакторы vi и Vim / А. Роббинс, Э. Хана, Л. Лэмб. – М.: Символ-Плюс, 2013. – 512 с.

2. Чан У. Дж. Python. Создание приложений. - М.: Вильямс, 2013. - 816 с.

3. Horstmann, C., Cornell, G. Core Java Volume I – Fundamentals (9th Edition) / C. Horstmann, G. Cornell. – Prentice Hall, 2012. – 1008 p.

4. Friedl, J.E.F. Mastering Regular Expressions / J.E.F. Friedl. – 3rd. ed. O'Reilly Media, 2012. – 544 p.

Методическая литература

1. Методические рекомендации к самостоятельной работе студентов по дисциплине «Основы программирования лингвистических алгоритмов». Каменский М.В. – Ставрополь, 2026

Интернет-ресурсы

1. Annotation Graph Toolkit (AGTK) – http://agtk.sourceforge.net
2. Applied Linguistics – http://www.appliedlinguistics.org
3. Center for Applied Linguistics – http://www.cal.org
4. Code::Blocks C/C++ IDE – http://www.codeblocks.org
5. Emacs – http://www.gnu.org/software/emacs
6. GATE (General Architecture for Text Engineering) – http://gate.ac.uk
7. GNU Compiler Collection – http://gcc.gnu.org
8. Helsinki Finite-State Technology – http://sourceforge.net/projects/hfst
9. Leopar Language Parser – http://leopar.loria.fr
10. NLTK (Natural Language Toolkit) – http://www.nltk.org
11. Open Natural Language Processing (OpenNLP) – http://opennlp.sourceforge.net
12. Praat – http://www.fon.hum.uva.nl/praat
13. Python Programming Language – http://www.python.org
14. Sed – http://www.gnu.org/s/sed
15. Sonic Visualizer – http://www.sonicvisualiser.org
16. **Vi Improved - http://www.vim.org**

Практическое занятие 4.
Тема: Строки в формате Unicode.
Цель: освоение студентами базовых принципов программирования алгоритмов обработки текстов на естественных языках с применением языка программирования Python.
Реализуемые компетенции:
ПК-4 - Способен применять лингводидактические знания при решении исследовательских, педагогических и прикладных задач, комплексно работать с лингвистической информацией в педагогической деятельности.
Актуальность: актуальность изучения темы определяется практической целесообразностью использования лингвистических алгоритмов для оптимизации деятельностиа лингвиста-практика в части обработки текстовой информации на естественных языках.

Вопросы и задания для собеседования (по материалам: URL: http://pep8.ru/doc/tutorial-3.1/).
Выполните следующие действия и задания.

О Unicode

Начиная с Python версии 3.0, строковый тип поддерживает только Unicode (см. http://www.unicode.org/).

Преимущество набора Unicode состоит в том, что он предоставляет порядковый номер для любого символа из любой письменности, использовавшейся в современных или древнейших текстах. До этих пор для символов в сценарии было доступно лишь 256 номеров. Тексты обычно привязывались к кодовой странице, которая устанавливала в соответствие порядковые номера и символы сценария. Это приводило к серьезной путанице, особенно в том, что касалось интернационализации программного продукта. Unicode решает эти проблемы, определяя единую кодовую страницу для всех письменностей.

Для вставки в строку специального символа можно использовать Unicode-экранирование (Python Unicode-Escape encoding). Следующий пример всё пояснит:

```
>>> 'Hello\u0020World !'
'Hello World !'
```

Экранированная последовательность \u0020 задаёт символ Unicode с порядковым номером 0x0020 (символ пробела).

Другие символы интерпретируются с использованием соответствующих им порядковых значений тем же способом, что и порядковые номера Unicode. Первые 128 символов кодировки Unicode полностью совпадают с 128 символами кодировки Latin-1, использующейся во многих западных странах.

Помимо стандартных способов кодирования, Python предоставляет целый набор различных способов создания Unicode-строк, основываясь на известной кодировке.

Для конвертирования Unicode-строки в последовательность байтов с использованием желаемой кодировки, строковые объекты предоставляют метод encode(), принимающий единственный параметр — название кодировки. Предпочитаются названия кодировок, записанные в нижнем регистре.

```
>>>"Äpfel".encode('utf-8')
b'\xc3\x84pfel'
```

Списки

В языке Python доступно некоторое количество *составных* типов данных, использующихся для группировки прочих значений вместе. Наиболее гибкий из них — список (list). Его можно выразить в тексте программы через разделённые запятыми значения (*элементы*), заключённые в квадратные скобки. Элементы списка могут быть разных типов.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Подобно индексам в строках, индексы списков начинаются с нуля, списки могут быть срезаны, объединены (*конкатенированы*) и так далее:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

В отличие от строк, являющихся неизменяемыми, изменить индивидуальные элементы списка вполне возможно:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Присваивание срезу также возможно, и это действие может даже изменить размер списка или полностью его очистить:

```
>>> # Заменим некоторые элементы:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Удалим немного:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Вставим пару:
... a[1:1] = ['bletch', 'xyzzy']
```

```
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Вставим (копию) самого себя в начало
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Очистка списка: замена всех значений пустым списком
>>> a[:] = []
>>> a
[]
```

Встроенная функция len() также применима к спискам:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

Вы можете встраивать списки (создавать списки, содержащие другие списки), например так:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
```

Вы можете добавить что-нибудь в конец списка.

```
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Обратите внимание, что в последнем примере p[1] и q на самом деле ссылаются на один и тот же объект! Мы вернёмся к *семантике объектов* позже.

Список рекомендуемой литературы

Основная литература

1. Саммерфилд М. Python на практике. М.: ДМП Пресс, 2014. - 338 с.

Дополительная литература

1. Роббинс, А. Изучаем редакторы vi и Vim / А. Роббинс, Э. Хана, Л. Лэмб. – М.: Символ-Плюс, 2013. – 512 с.
2. Чан У. Дж. Python. Создание приложений. - М.: Вильямс, 2013. - 816 с.
3. Horstmann, C., Cornell, G. Core Java Volume I – Fundamentals (9th Edition) / C. Horstmann, G. Cornell. – Prentice Hall, 2012. – 1008 p.

4. Friedl, J.E.F. Mastering Regular Expressions / J.E.F. Friedl. – 3rd. ed. O'Reilly Media, 2012. – 544 p.

Методическая литература

1. Методические рекомендации к самостоятельной работе студентов по дисциплине «Основы программирования лингвистических алгоритмов». Каменский М.В. – Ставрополь, 2026

Интернет-ресурсы

1. Annotation Graph Toolkit (AGTK) – http://agtk.sourceforge.net
2. Applied Linguistics – http://www.appliedlinguistics.org
3. Center for Applied Linguistics – http://www.cal.org
4. Code::Blocks C/C++ IDE – http://www.codeblocks.org
5. Emacs – http://www.gnu.org/software/emacs
6. GATE (General Architecture for Text Engineering) – http://gate.ac.uk
7. GNU Compiler Collection – http://gcc.gnu.org
8. Helsinki Finite-State Technology – http://sourceforge.net/projects/hfst
9. Leopar Language Parser – http://leopar.loria.fr
10. NLTK (Natural Language Toolkit) – http://www.nltk.org
11. Open Natural Language Processing (OpenNLP) – http://opennlp.sourceforge.net
12. Praat – http://www.fon.hum.uva.nl/praat
13. Python Programming Language – http://www.python.org
14. Sed – http://www.gnu.org/s/sed
15. Sonic Visualizer – http://www.sonicvisualiser.org
16. **Vi Improved - http://www.vim.org**

Практическое занятие 5.
Тема: Базовые приемы программирования на ЯП Python
Цель: освоение студентами базовых принципов программирования алгоритмов обработки текстов на естественных языках с применением языка программирования Python.
Реализуемые компетенции:
ПК-4 - Способен применять лингводидактические знания при решении исследовательских, педагогических и прикладных задач, комплексно работать с лингвистической информацией в педагогической деятельности.
Актуальность: актуальность изучения темы определяется практической целесообразностью использования лингвистических алгоритмов для оптимизации деятельности лингвиста-практика в части обработки текстовой информации на естественных языках.

Вопросы и задания для собеседования (по материалам: URL: http://pep8.ru/doc/tutorial-3.1/).
Выполните следующие задания и действия.

Безусловно, Python можно использовать для более сложных задач, чем сложение двух чисел. Например, мы можем вывести начало последовательности чисел Фибоначчи таким образом:

```
>>> # Ряд Фибоначчи:
... # сумма двух элементов определяет следущий элемент
... a, b = 0, 1
>>> while b < 10:
... print(b)
... a, b = b, a+b
...
1
1
2
3
5
8
```

Этот пример показывает нам некоторые новые возможности.

1. Первая строка содержит *множественное присваивание* (multiple assignment): переменные a и b параллельно получают новые значения — 0 и 1. В последней строке этот метод используется снова, демонстрируя тот факт, что выражения по правую сторону [от оператора присваивания] всегда вычисляются раньше каких бы то ни было присваиваний. Сами же разделённые запятыми выражения вычисляются слева направо.

- Цикл while (пока) исполняется до тех пор, пока условие (здесь: b < 10) остается истиной. В Python, также как и в C, любое ненулевое значение является истиной (True); ноль является ложью (False). Условием может быть строка, список или вообще любая последовательность; все, что имеет ненулевую длину, играет роль истины, пустые последовательности — лжи. Использованная в примере проверка — простое условие. Стандартные операции сравнения записываются так же, как и в C: < (меньше чем), > (больше чем), == (равно), <= (меньше или равно), >= (больше или равно) и != (не равно).

- *Тело* цикла выделено *отступом* (indented). Отступы — это средство группировки операторов в Python. Интерактивный режим Python (пока!) не имеет какого-либо

разумного и удобного средства для редактирования строк ввода, поэтому необходимо использовать табуляции или пробелы для отступа в каждой строке. На практике более сложный текст на Python готовится в текстовом редакторе, а большинство из них имеют функцию авто-отступа. По окончанию ввода составного выражения в интерактивном режиме, необходимо закончить его пустой строкой — признаком завершения (поскольку интерпретатор не может угадать, когда вами была введена последняя строка). Обратите внимание, что размер отступа в каждой строке основного блока должен быть одним и тем же.

- Функция print() выводит значения переданных ей выражений. Поведение этой функции отличается от обычного вывода выражения (как происходило выше в примерах с калькулятором) тем, каким способом обрабатываются ряды выражений, величины с плавающей точкой и строки. Строки выводятся без кавычек и между элементами вставляются пробелы, благодаря чему форматирование вывода улучшается — как, например, здесь:

```
>>> i = 256*256
>>> print('Значением i является', i)
Значением i является 65536
```

Для отключения перевода строки после вывода или завершения вывода другой строкой используется именованный параметр end:

```
>>> a, b = 0, 1
>>> while b < 1000:
... print(b, end=' ')
... a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Больше средств для управления потоком команд

Помимо описанного выше оператора while, в Python доступны привычные операторы управления потоком из других языков, но с некоторыми особенностями.

Оператор if

Возможно, наиболее широко известный тип оператора — оператор if (если). Пример:

```
>>> x = int(input("Введите, пожалуйста, целое число: "))
Введите, пожалуйста, целое число: 42
>>> if x < 0:
... x = 0
... print('Отрицательное значение, изменено на ноль')
... elif x == 0:
... print('Ноль')
... elif x == 1:
... print('Один')
... else:
... print('Больше')
...
```

Блока elif может не быть вообще, он может быть один или их может быть несколько, а блок else (иначе) необязателен. Ключевое слово elif — краткая запись else if (иначе если) — позволяет избавиться от чрезмерного количества отступов. Оператор if ... elif ... elif ... — аналог оператора выбора switch или case, которые можно встретить в других языках программирования.

Оператор for

Оператор for в Python немного отличается от того, какой вы, возможно, использовали в C или Pascal. Вместо неизменного прохождения по арифметической прогрессии из чисел (как в Pascal) или предоставления пользователю возможности указать шаг итерации и условие остановки (как в C), оператор for в Python проходит по всем элементам любой последовательности (списка или строки) в том порядке, в котором они в ней располагаются. Например (игра слов не подразумевалась):

```
>>> # Измерим несколько строк:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
... print(x, len(x))
...
cat 3
window 6
defenestrate 12
```

Изменять содержимое последовательности, по которой проходит цикл, небезопасно (это в принципе-то возможно только для изменяемых типов, таких как списки). Если необходимо модифицировать список, использующийся для организации цикла, (например, для того чтобы продублировать отдельные элементы) нужно передать циклу его копию. Нотация срезов делает это практически безболезненным:

```
>>> for x in a[:]: # создать срез-копию всего списка
... if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

Функция range()

Если вам нужно перебрать последовательность чисел, встроенная функция range() придёт на помощь. Она генерирует арифметические прогрессии:

```
>>> for i in range(5):
... print(i)
...
0
1
2
3
4
```

Указанный конец интервала никогда не включается в сгенерированный список; вызов range(10) генерирует десять значений, которые являются подходящими индексами для элементов

последовательности длины 10. Можно указать другое начало интервала и другую, даже отрицательную, величину шага.

range(5, 10)
от 5 до 9

range(0, 10, 3)
0, 3, 6, 9

range(-10, -100, -30)
-10, -40, -70

Чтобы пройти по всем индексам какой-либо последовательности, скомбинируйте вызовы range() и len() следующим образом:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
... print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

В большинстве таких случаев удобно использовать функцию enumerate(), обратитесь к Организация циклов.

Странные вещи начинают происходить при попытке вывода последовательности:

```
>>> print(range(10))
range(0, 10)
```

Во многих случаях объект, возвращаемый функцией range(), ведёт себя как список, но фактически им не является. Этот объект возвращает по очереди элементы желаемой последовательности, когда вы проходите по нему в цикле, но на самом деле не создаёт списка, сохраняя таким образом пространство в памяти.

Мы называем такие объекты *итерируемыми* (iterable), и это все объекты, которые предназначаются для функций и конструкций, ожидающих от них поочерёдного предоставления элементов до тех пор, пока источник не иссякнет. Мы видели, что оператор for является таким *итератором* iterator. Функция list() тоже из их числа — она создаёт списки из *итерируемых* объектов:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Позже мы познакомимся и с другими функциями, которые возвращают и принимают *итерируемые* объекты в качестве параметров.

Операторы break и continue, а также условие else в циклах

Оператор break прерывает выполнение самого ближайшего вложенного цикла for или while (по аналогии с языком C).

Оператор continue, также заимствованный из C, продолжает выполнение цикла со следующей итерации.

Операторы циклов могут иметь ветвь else. Она исполняется, когда цикл выполнил перебор до конца (в случае for) или когда условие становится ложным (в случае while), но не в тех случаях, когда цикл прерывается по break. Это поведение иллюстрируется следующим примером, в котором производится поиск простых чисел:

```
>>> for n in range(2, 10):
... for x in range(2, int(n ** 0.5) + 1):
... if n % x == 0:
... print(n, 'равно', x, '*', n//x)
... break
... else:
... print(n, '- простое число')
...
2 - простое число
3 - простое число
4 равно 2 * 2
5 - простое число
6 равно 2 * 3
7 - простое число
8 равно 2 * 4
9 равно 3 * 3
```

Оператор pass

Оператор pass не делает ничего. Он может использоваться когда синтаксически требуется присутствие оператора, но от программы не требуется действий. Например:

```
>>> while True:
... pass # Ожидание прерывания с клавиатуры (Ctrl+C) в режиме занятости
...
```

Этот оператор часто используется для создания минималистичных классов, к примеру *исключений* (exceptions), или для игнорирования нежелательных исключений:

```
>>> class ParserError(Exception):
... pass
...
>>> try:
... import audioop
... except ImportError:
... pass
...
```

Другой вариант: pass может применяться в качестве заглушки для тела функции или условия при создании нового кода, позволяя вам сохранить абстрактный взгляд на вещи. С другой стороны, оператор pass игнорируется без каких-либо сигналов и лучшим выбором было бы породить исключение NotImplementedError:

```
>>> def initlog(*args):
... raise NotImplementedError  # Открыть файл для логгинга, если он ещё не открыт
... if not logfp:
... raise NotImplementedError  # Настроить заглушку для логгинга
... raise NotImplementedError('Обработчик инициализации лога вызовов')
...
```

Если бы здесь использовались операторы pass, а позже вы бы запускали тесты, они могли бы упасть без указания причины. Использование NotImplementedError принуждает этот код породить исключение, сообщая вам конкретное место, где присутствует незавершённый код. Обратите внимание на два способа порождения исключений. Первый способ, без сообщения, но сопровождаемый комментарием, позволяет вам оставить комментарий когда вы будете подменять выброс исключения рабочим кодом, который, в свою очередь, в идеале, будет хорошим описанием блока кода, для которого исключение предназначалось заглушкой. Однако, передача сообщения вместе с исключением, как в третьем примере, обуславливает более насыщенный информацией вывод при отслеживании ошибки.

Список рекомендуемой литературы

Основная литература

1. Саммерфилд М. Python на практике. М.: ДМП Пресс, 2014. - 338 с.

Дополительная литература

1. Роббинс, А. Изучаем редакторы vi и Vim / А. Роббинс, Э. Хана, Л. Лэмб. – М.: Символ-Плюс, 2013. – 512 с.

2. Чан У. Дж. Python. Создание приложений. - М.: Вильямс, 2013. - 816 с.

3. Horstmann, C., Cornell, G. Core Java Volume I – Fundamentals (9th Edition) / C. Horstmann, G. Cornell. – Prentice Hall, 2012. – 1008 p.

4. Friedl, J.E.F. Mastering Regular Expressions / J.E.F. Friedl. – 3rd. ed. O'Reilly Media, 2012. – 544 p.

Методическая литература

1. Методические рекомендации к самостоятельной работе студентов по дисциплине «Основы программирования лингвистических алгоритмов». Каменский М.В. – Ставрополь, 2026

Интернет-ресурсы

1. Annotation Graph Toolkit (AGTK) – http://agtk.sourceforge.net
2. Applied Linguistics – http://www.appliedlinguistics.org
3. Center for Applied Linguistics – http://www.cal.org
4. Code::Blocks C/C++ IDE – http://www.codeblocks.org
5. Emacs – http://www.gnu.org/software/emacs
6. GATE (General Architecture for Text Engineering) – http://gate.ac.uk
7. GNU Compiler Collection – http://gcc.gnu.org
8. Helsinki Finite-State Technology – http://sourceforge.net/projects/hfst
9. Leopar Language Parser – http://leopar.loria.fr
10. NLTK (Natural Language Toolkit) – http://www.nltk.org
11. Open Natural Language Processing (OpenNLP) – http://opennlp.sourceforge.net

12. Praat – http://www.fon.hum.uva.nl/praat
13. Python Programming Language – http://www.python.org
14. Sed – http://www.gnu.org/s/sed
15. Sonic Visualizer – http://www.sonicvisualiser.org
16. **Vi Improved - http://www.vim.org**

Практическое занятие 6.
Тема: Определение функций в Python
Цель: освоение студентами базовых принципов программирования алгоритмов обработки текстов на естественных языках с применением языка программирования Python.
Реализуемые компетенции:
ПК-4 - Способен применять лингводидактические знания при решении исследовательских, педагогических и прикладных задач, комплексно работать с лингвистической информацией в педагогической деятельности.
Актуальность: актуальность изучения темы определяется практической целесообразностью использования лингвистических алгоритмов для оптимизации деятельности лингвиста-практика в части обработки текстовой информации на естественных языках.

Вопросы и задания для собеседования (по материалам: URL: http://pep8.ru/doc/tutorial-3.1/).
Выполните следующие действия и задания.

Мы можем создать функцию, которая выводит числа Фибоначчи до некоторого предела:

```
>>> def fib(n): # вывести числа Фибоначчи меньшие (вплоть до) n
... """Выводит ряд Фибоначчи, ограниченный n."""
... a, b = 0, 1
... while b < n:
... print(b, end=' ')
... a, b = b, a+b
...
>>> # Теперь вызовем определенную нами функцию:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Зарезервированное слово def предваряет *определение* функции. За ним должны следовать имя функции и заключённый в скобки список формальных параметров. Выражения, формирующие тело функции, начинаются со следующей строки и должны иметь отступ.

Первым выражением в теле функции может быть строковой литерал — этот литерал является строкой документации функции, или *док-строкой* (docstring). (Больше информации о *док-строках* вы найдёте в разделе Строки документации) Существуют инструменты, которые используют *док-строки* для того, чтобы сгенерировать печатную или онлайн-документацию или чтобы позволить пользователю перемещаться по коду интерактивно; добавление строк документации в ваш код — это хорошая практика, постарайтесь к ней привыкнуть.

Исполнение функции приводит к созданию новой таблицы символов, использующейся для хранения локальных переменных функции. Если быть более точными, все присваивания переменных в функции сохраняют значение в локальной таблице символов; при обнаружении ссылки на переменную, в первую очередь просматривается локальная таблица символов, затем локальная таблица символов для окружающих функций, затем глобальная таблица символов и, наконец, таблица встроенных имён. Таким образом, глобальным переменным невозможно прямо присвоить значения внутри функций (если они конечно не упомянуты в операторе global) несмотря на то, что ссылки на них могут использоваться.

Фактические параметры при вызове функции помещаются в локальную таблицу символов вызванной функции; в результате аргументы передаются через *вызов по значению* (call by value)

(где значение — это всегда *ссылка* (reference) на объект, а не значение его самого). Если одна функция вызывает другую — то для этого вызова создается новая локальная таблица символов.

При определении функции её имя также помещается в текущую таблицу символов. Тип значения, связанного с именем функции, распознается интерпретатором как функция, определённая пользователем (user-defined function). Само значение может быть присвоено другому имени, которое затем может также использоваться в качестве функции. Эта система работает в виде основного механизма переименования:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Если вы использовали в работе другие языки программирования, вы можете возразить, что fib — это не функция, а процедура, поскольку не возвращает никакого значения. На самом деле, даже функции без ключевого слова return возвращают значение, хотя и более скучное. Такое значение именуется None (это встроенное имя). Вывод значения None обычно подавляется в интерактивном режиме интерпретатора, если оно оказывается единственным значением, которое нужно вывести. Вы можете проследить за этим процессом, если действительно хотите, используя функцию print():

```
>>> fib(0)
>>> print(fib(0))
None
```

Довольно легко написать функцию, которая возвращает список чисел из ряда Фибоначчи, вместо того, чтобы выводить их:

```
>>> def fib2(n): # вернуть числа Фибоначчи меньшие (вплоть до) n
... """Возвращает список чисел ряда Фибоначчи, ограниченный n."""
... result = []
... a, b = 0, 1
... while b < n:
... result.append(b) # см. ниже
... a, b = b, a+b
... return result
...
>>> f100 = fib2(100) # вызываем
>>> f100 # выводим результат
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

И на этот раз пример демонстрирует некоторые новые возможности Python:

- Оператор return завершает выполнение функции, возвращая некоторое значение. Оператор return без аргумента возвращает None. Достижение конца функции также возвращает None.

- Выражение result.append(b) вызывает метод append объекта-списка result. Метод — это функция, которая «принадлежит» объекту и указывается через выражение вида

obj.methodname, где obj — некоторый объект (может быть выражением), a methodname — имя метода, присущий объекту данного типа. Различные типы определяют различные методы. Методы разных типов могут иметь одинаковые имена, не вызывая неопределённостей. (Позже в этом учебнике будет рассмотрено как определять собственные типы объектов и методы, используя классы.) Метод append(), показанный в примере, определён для объектов типа список. Он добавляет в конец списка новый элемент. В данном примере это действие эквивалентно выражению result = result + [b], но более эффективно.

Список рекомендуемой литературы

Основная литература

1. Саммерфилд М. Python на практике. М.: ДМП Пресс, 2014. - 338 с.

Дополительная литература

1. Роббинс, А. Изучаем редакторы vi и Vim / А. Роббинс, Э. Хана, Л. Лэмб. – М.: Символ-Плюс, 2013. – 512 с.

2. Чан У. Дж. Python. Создание приложений. - М.: Вильямс, 2013. - 816 с.

3. Horstmann, C., Cornell, G. Core Java Volume I – Fundamentals (9th Edition) / C. Horstmann, G. Cornell. – Prentice Hall, 2012. – 1008 p.

4. Friedl, J.E.F. Mastering Regular Expressions / J.E.F. Friedl. – 3rd. ed. O'Reilly Media, 2012. – 544 p.

Методическая литература

1. Методические рекомендации к самостоятельной работе студентов по дисциплине «Основы программирования лингвистических алгоритмов». Каменский М.В. – Ставрополь, 2026

Интернет-ресурсы

1. Annotation Graph Toolkit (AGTK) – http://agtk.sourceforge.net
2. Applied Linguistics – http://www.appliedlinguistics.org
3. Center for Applied Linguistics – http://www.cal.org
4. Code::Blocks C/C++ IDE – http://www.codeblocks.org
5. Emacs – http://www.gnu.org/software/emacs
6. GATE (General Architecture for Text Engineering) – http://gate.ac.uk
7. GNU Compiler Collection – http://gcc.gnu.org
8. Helsinki Finite-State Technology – http://sourceforge.net/projects/hfst
9. Leopar Language Parser – http://leopar.loria.fr
10. NLTK (Natural Language Toolkit) – http://www.nltk.org
11. Open Natural Language Processing (OpenNLP) – http://opennlp.sourceforge.net
12. Praat – http://www.fon.hum.uva.nl/praat
13. Python Programming Language – http://www.python.org
14. Sed – http://www.gnu.org/s/sed
15. Sonic Visualizer – http://www.sonicvisualiser.org
16. **Vi Improved - http://www.vim.org**

Практическое занятие 7.
Тема: Определение функций в Python: сложные конструкции определения функций
Цель: освоение студентами базовых принципов программирования алгоритмов обработки текстов на естественных языках с применением языка программирования Python.
Реализуемые компетенции:
ПК-4 - Способен применять лингводидактические знания при решении исследовательских, педагогических и прикладных задач, комплексно работать с лингвистической информацией в педагогической деятельности.
Актуальность: актуальность изучения темы определяется практической целесообразностью использования лингвистических алгоритмов для оптимизации деятельности лингвиста-практика в части обработки текстовой информации на естественных языках.

Вопросы и задания для собеседования (по материалам: URL: http://pep8.ru/doc/tutorial-3.1/).
Выполните следующие действия и задания.

Также есть возможность определять функции с переменным количеством параметров. Для этого существует три формы, которые также можно использовать совместно.

Значения аргументов по умолчанию

Наиболее полезной формой является задание значений по умолчанию для одного или более параметров. Таким образом создаётся функция, которая может быть вызвана с меньшим количеством параметров, чем в её определении: при этом неуказанные при вызове параметры примут данные в определении функции значения. Например:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
while True:
ok = input(prompt)
if ok in ('y', 'yeah', 'yes', 'yep'): return True
if ok in ('n', 'no', 'nop', 'nope'): return False
retries = retries - 1
if retries < 0:
raise IOError('refusenik user')
print(complaint)
```

Эта функция может быть вызвана, например, так: ask_ok('Do you really want to quit?') или так: ask_ok('OK to overwrite the file?', 2).

Этот пример также знакомит вас с зарезервированным словом in. Посредством его можно проверить, содержит ли последовательность определённое значение или нет.

Значения по умолчанию вычисляются в месте определения функции, в *определяющей* области, поэтому код

```
i = 5

def f(arg=i):
print(arg)

i = 6
f()
```

выведет 5.

Важное предупреждение: Значение по умолчанию вычисляется лишь единожды. Это особенно важно помнить, когда значением по умолчанию является изменяемый объект, такой как список, словарь (dictionary) или экземпляры большинства классов. Например, следующая функция накапливает переданные ей параметры:

```
def f(a, L=[]):
L.append(a)
return L

print(f(1))
print(f(2))
print(f(3))
```

Она выведет

```
[1]
[1, 2]
[1, 2, 3]
```

Если вы не хотите, чтобы значение по умолчанию распределялось между последовательными вызовами, вместо предыдущего варианта вы можете использовать такую идиому:

```
def f(a, L=None):
if L is None:
L = []
L.append(a)
return L
```

Именованные параметры

Функции также могут быть вызваны с использованием именованных параметров (keyword arguments) в форме «*имя = значение*». Например, нижеприведённая функция:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
print("-- This parrot wouldn't", action, end=' ')
print("if you put", voltage, "volts through it.")
print("-- Lovely plumage, the", type)
print("-- It's", state, "!")
```

могла бы быть вызвана любым из следующих способов:

```
parrot(1000)
parrot(action='VOOOOOM', voltage=1000000)
parrot('a thousand', state='pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

а эти случаи оказались бы неверными:

```
parrot() # пропущен требуемый аргумент
parrot(voltage=5.0, 'dead') # позиционный параметр вслед за именованным
```

```
parrot(110, voltage=220)  # повторное значение параметра
parrot(actor='John Cleese') # неизвестное имя параметра
```

В общем случае, список параметров должен содержать любое количество позиционных (positional) параметров, за которыми может следовать любое количество именованных, и при этом имена аргументов выбираются из формальных параметров. Неважно, имеет формальный параметр значение по умолчанию или нет. Ни один из аргументов не может получать значение более чем один раз — имена формальных параметров, совпадающие с именами позиционных параметров, не могут использоваться в качестве именующих в одном и том же вызове. Вот пример, завершающийся неудачей по причине описанного ограничения:

```
>>> def function(a):
... pass
...
>>> function(0, a=0)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Если в определении функции присутствует завершающий параметр в виде ***имя*, он получит в качестве значения словарь (подробнее в разделе Справочника — Типы-отображения — словари), содержащий все именованные параметры и их значения, исключая те, которые соответствуют формальным параметрам. Можно совместить эту особенность с поддержкой формального параметра в формате **имя* (описывается в следующем подразделе), который получает кортеж (tuple), содержащий все позиционные параметры, следующие за списком формальных параметров. (параметр в формате **имя* должен описываться перед параметром в формате ***имя*.) Например, если мы определим такую функцию:

```
def cheeseshop(kind, *arguments, **keywords):
print("-- Do you have any", kind, "?")
print("-- I'm sorry, we're all out of", kind)
for arg in arguments: print(arg)
print("-" * 40)
keys = sorted(keywords.keys())
for kw in keys: print(kw, ":", keywords[kw])
```

то её можно будет вызвать так:

```
cheeseshop('Limburger', "It's very runny, sir.",
"It's really very, VERY runny, sir.",
client="John Cleese",
shopkeeper="Michael Palin",
sketch="Cheese Shop Sketch")
```

и она, конечно же, выведет:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
```

client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch

Обратите внимание, что список имён (ключей) именованных параметров (keys) создается посредством сортировки содержимого списка ключей keys() словаря keywords; если бы этого не было сделано, порядок вывода параметров был бы произволен.

Списки параметров произвольной длины

Наконец, наиболее редко используется возможность указания того, что функция может быть вызвана с произвольным числом аргументов. При этом сами параметры будут обёрнуты в кортеж (см. раздел Кортежи). Переменное количество параметров могут предварять ноль или более обычных.

```
def write_multiple_items(file, separator, *args):
file.write(separator.join(args))
```

Обычно параметры неизвестного заранее количества (variadic) указываются последними в списке формальных параметров, поскольку включают в себя все остальные переданные в функцию параметры. Все формальные параметры, которые следуют за параметром *args, должны быть только именованными, то есть, они могут быть заданы только по имени (в отличие от позиционных параметров).

```
>>> def concat(*args, sep="/"):
... return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

Распаковка списков параметров

Обратная ситуация возникает когда параметры уже содержатся в списке или в кортеже, но должны быть распакованы для вызова функции, требующей отдельных позиционных параметров. Например, встроенная функция range() ожидает отдельные параметры *start* и *stop* соответственно. Если они не доступны раздельно, для распаковки аргументов из списка или кортежа в вызове функции используйте *-синтаксис:

```
>>> list(range(3, 6)) # обычный вызов с отдельными параметрами
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args)) # вызов с распакованными из списка параметрами
[3, 4, 5]
```

Схожим способом, словари могут получать именованные параметры через **-синтаксис:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
... print("-- This parrot wouldn't", action, end=' ')
... print("if you put", voltage, "volts through it.", end=' ')
... print("It's", state, "!")
```

```
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. It's bleedin' demised !
```

Модель lambda

В связи с неустанными просьбами, в Python были добавлены несколько возможностей, которые были привычны для функциональных языков программирования, таких как Lisp. Используя зарезервированное слово lambda, вы можете создать небольшую безымянную функцию. Например, функцию, которая возвращает сумму двух своих аргументов, можно записать так: lambda a, b: a+b. Формы lambda могут быть использованы в любом месте где требуется объект функции. При этом они синтаксически ограничены одним выражением. Семантически, они лишь «синтаксический сахар» для обычного определения функции. Как и определения вложенных функций, lambda-формы могут ссылаться на переменные из содержащей их области видимости:

```
>>> def make_incrementor(n):
... return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Строки документации

Перечислим некоторые существующие соглашения по содержимому строк документации и их форматированию.

По поводу форматирования строк документации и их содержимого постоянно появляются всё новые соглашения.

Первая строка всегда должна быть сжатой, лаконичной сводкой о назначении объекта. Для краткости, в ней не обязательно присутствие имени типа или объекта, поскольку они доступны другими способами (исключая случай, когда имя функции оказывается глаголом, описывающим суть операции). Эта строка должна начинаться с прописной буквы и оканчиваться точкой.

Если строке документации (литералу, объекту строки) требуется больше строк (физических), вторая строка должна быть пустой, визуально отделяя сводку от остального описания. Следующие строки могут быть одним или более абзацем, описывающим соглашения по вызову объекта, сторонние эффекты, и т. д.

Парсер Python не обрабатывает отступы в много-строковых литералах, поэтому инструментам, которые работают над документацией, предлагается, по желанию, делать это самим. Производится это по следующему соглашению. Первая непустая строка после первой строки литерала определяет величину отступа всего литерала документации. (Мы не можем использовать первую строку, поскольку она обычно выравнивается по открывающим кавычкам и её отступ в литерале не явен). Пробельный «эквивалент» этого отступа затем отрезается от начала всех строк литерала. Строк с меньшим отступом не должно обнаруживаться, но если они встретились, весь их начальный отступ должен быть обрезан. Эквивалентность пробельных

замен может быть протестирована развертыванием табуляции (обычно, к 8 пробелам).

Вот пример многострочной документации (*док-строки*):

```
>>> def my_function():
... """Не делаем ничего, но документируем.
...
... Нет, правда, эта функция ничего не делает.
... """
... pass
...
>>> print(my_function.__doc__)
Не делаем ничего, но документируем.
```

Список рекомендуемой литературы

Основная литература

1. Саммерфилд М. Python на практике. М.: ДМП Пресс, 2014. - 338 с.

Дополительная литература

1. Роббинс, А. Изучаем редакторы vi и Vim / А. Роббинс, Э. Хана, Л. Лэмб. – М.: Символ-Плюс, 2013. – 512 с.
2. Чан У. Дж. Python. Создание приложений. - М.: Вильямс, 2013. - 816 с.
3. Horstmann, C., Cornell, G. Core Java Volume I – Fundamentals (9th Edition) / C. Horstmann, G. Cornell. – Prentice Hall, 2012. – 1008 p.
4. Friedl, J.E.F. Mastering Regular Expressions / J.E.F. Friedl. – 3rd. ed. O'Reilly Media, 2012. – 544 p.

Методическая литература

1. Методические рекомендации к самостоятельной работе студентов по дисциплине «Основы программирования лингвистических алгоритмов». Каменский М.В. – Ставрополь, 2026

Интернет-ресурсы

1. Annotation Graph Toolkit (AGTK) – http://agtk.sourceforge.net
2. Applied Linguistics – http://www.appliedlinguistics.org
3. Center for Applied Linguistics – http://www.cal.org
4. Code::Blocks C/C++ IDE – http://www.codeblocks.org
5. Emacs – http://www.gnu.org/software/emacs
6. GATE (General Architecture for Text Engineering) – http://gate.ac.uk
7. GNU Compiler Collection – http://gcc.gnu.org
8. Helsinki Finite-State Technology – http://sourceforge.net/projects/hfst
9. Leopar Language Parser – http://leopar.loria.fr
10. NLTK (Natural Language Toolkit) – http://www.nltk.org
11. Open Natural Language Processing (OpenNLP) – http://opennlp.sourceforge.net
12. Praat – http://www.fon.hum.uva.nl/praat
13. Python Programming Language – http://www.python.org
14. Sed – http://www.gnu.org/s/sed
15. Sonic Visualizer – http://www.sonicvisualiser.org
16. Vi Improved - http://www.vim.org

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания

по выполнению самостоятельной работы студентов по дисциплине
«Основы лингвистического программирования»

Направление подготовки	45.04.02 Лингвистика
Направленность (профиль)	Современные методы прикладной лингвистики и перевода
Год начала подготовки	2026
Форма обучения	Очная
Реализуется в семестре	2

Ставрополь
2026

Содержание

1. Введение

Методические рекомендации к самостоятельной работе магистрантов по дисциплине «Основы лингвистического программирования» разработаны в соответствии с рабочей программой дисциплины по направлению 45.04.02 - Лингвистика, программа – Современные методы прикладной лингвистики и перевода.

Основной формой работы магистранта является не только работа на лекции, изучение конспекта лекций, их дополнение рекомендованной литературой, но и большая самостоятельная учебная работа, которая позволит глубоко проникнуть в суть рассматриваемой проблемы и подготовить почву для написания кандидатской диссертации. Но для успешной учебной деятельности, ее интенсификации необходимо учитывать следующие субъективные факторы:

1. Знание программного материала, наличие прочной системы знаний, необходимой для усвоения основных дисциплин, предусмотренных программой, общая совокупность которых обусловливает уровень овладения грамматическим компонентом иноязычной речи.

2. Наличие выработанных умений, навыков умственного труда:

а) умение делать глубокий, обстоятельный анализ при работе с книгой, Интернет–источниками;

б) владение логическими операциями: сравнение, анализ, обобщение, определение понятий, правила систематизации и классификации.

3. Специфика познавательных психических процессов: внимание, память, речь, наблюдательность, интеллект и мышление.

4. Хорошая работоспособность, которая обеспечивается нормальным физическим состоянием.

5. Соответствие избранной деятельности, профессии индивидуальным способностям. Необходимо выработать умение саморегулировать свое эмоциональное состояние и устранять обстоятельства, нарушающие деловой настрой, мешающие намеченной работе.

6. Овладение оптимальным стилем работы, обеспечивающим успех в деятельности.

7. Уровень требований к себе, определяемый сложившейся самооценкой.

Адекватная оценка знаний, достоинств, недостатков – важная составляющая самоорганизации человека, без нее невозможна успешная работа по управлению своим поведением, деятельностью.

По наблюдениям исследователей педагогов, одна из основных особенностей обучения заключается в том, что постоянный внешний контроль заменяется самоконтролем, активная роль в обучении принадлежит уже не столько преподавателю, сколько магистранту.

2. Общая характеристика самостоятельной работы студента при изучении дисциплины

Самостоятельная работа магистранта в рамках дисциплины «Основы программирования лингвистических алгоритмов» понимается как планируемая учебная работа, выполняемая во внеаудиторное (аудиторное) время по заданию и при методическом руководстве преподавателя, но без его непосредственного участия.

Самостоятельная работа направлена на формирование следующих компетенций:

Индекс	Формулировка:
ПК-4	Способен применять лингводидактические знания при решении исследовательских, педагогических и прикладных задач, комплексно работать с лингвистической информацией в педагогической деятельности.

Цель самостоятельной работы магистрантов в процессе изучения дисциплины «Основы лингвистического программирования» – научить магистранта осмысленно и самостоятельно работать: 1) с учебным материалом по дисциплине, 2) с научной информацией, актуальными исследованиями в области лингвистики, 3) с эмпирическими данными, получаемыми в ходе экспериментальных лингвистических исследований, 4) с методологическими подходами современных лингвистических исследований; 5) с конкретными лингвистическими методами и методиками.

Задачи самостоятельной работы:

- систематизировать и закрепить полученные теоретические знания и практические умения магистрантов;
- развить познавательные способности и активность магистрантов: творческую инициативу, самостоятельность, ответственность и организованность;
- сформировать и развить навыки ведения самостоятельной работы и овладения методикой исследования при решении разрабатываемых в учебной деятельности проблем и вопросов;
-повысить уровень подготовленности к самостоятельной работе в соответствии с выбранным научным направлением в условиях современного состояния науки и культуры.

Таким образом, самостоятельная работа приобщает научному и исследовательскому творчеству, поиску и анализу актуальных проблем современной психолингвистической науки.

3. План-график выполнения самостоятельной работы.

Код реализуемой компетенции	Вид деятельности студентов	Итоговый продукт самостоятельной работы	Средства и технологии оценки	Объем часов
ПК-4	Изучение литературы	Конспект	Собеседование	14,5
ПК-4	Подготовка к круглому столу	Индивидуальное задание	Собеседование	14,5
ПК-4	Работа с электронными ресурсами в сети Интернет	Конспект	Собеседование	14,5
ПК-4	Подготовка доклада	Доклад и презентация	Собеседование	14,5
Итого за 2 семестр				58
Итого				58

Для выполнения самостоятельной работы необходимо пользоваться литературой, которая предложена в списке рекомендуемой литературы, Интернет-ресурсами или другими источниками по усмотрению магистранта.

Самостоятельная работа рассчитана на разные уровни мыслительной деятельности. Выполненная работа позволит приобрести не только знания, но и умения, навыки, а также выработать свою методику подготовки, что очень важно в дальнейшем процессе научной деятельности.

При изучении дисциплины предусматриваются следующие формы самостоятельной работы магистранта:

- самостоятельное изучение основной и дополнительной литературы по дисциплине с конспектированием по разделам;
- работа с электронными ресурсами в сети Интернет;
- конспектирование и реферирование первоисточника и научно-исследовательской литературы;
- подготовка к семинару-круглому столу;
- подготовка мультемедийной презентации;
- подготовка доклада.

4. Методические рекомендации по изучению теоретического материала

Чтение основной и дополнительной литературы по курсу с конспектированием по разделам.

Самостоятельная работа при чтении учебной литературы начинается с изучения конспекта материала, полученного при слушании лекций преподавателя. Полученную информацию необходимо осмыслить. При необходимости, в конспект лекций могут быть внесены схемы, другая

дополнительная информация. При изучении нового материала составляется конспект. Сжато излагается самое существенное в данном материале.

Работа с электронными ресурсами в сети Интернет.

Для повышения эффективности самостоятельной работы магистрант должен уметь работать в поисковой системе сети Интернет и использовать найденную информацию при подготовке к занятиям. Поиск информации можно вести по автору, заглавию, виду издания, году издания или издательству. Также в сети Интернет доступна услуга по скачиванию методических указаний и учебных пособий, подбору необходимой научной литературы.

Конспектирование и реферирование первоисточника и научно-исследовательской литературы.

Конспект представляет собой дословные выписки из текста источника. При этом необходимо понимать, что конспект – это не полное переписывание чужого текста. Необходимо знать, что при написании конспекта сначала прочитывается текст – источник, в нём выделяются основные положения, подбираются примеры, идёт перекомпоновка материала, а уже затем оформляется текст конспекта. Конспект может быть полным, когда работа идёт со всем текстом источника или неполным, когда интерес представляет какой-либо один или несколько вопросов, затронутых в источнике.

Реферирование — это сложный творческий процесс, в основе которого лежит умение выделить главную информацию из текста первоисточника. Реферирование – процесс аналитически-синтетического обработки информации, которая заключается в анализе первичного документа, нахождении значимых в смысловом отношении данных (основных положений, фактов, доведите день, результатов, выводов) Реферирование имеет целью сократить физический объем первичного документа при сохранении его основного смыслового содержания, используется в научной, издательской, информационной и библиографической деятельности.

5 Методические указания (по видам работ, предусмотренных рабочей программой дисциплины)

Подготовка к круглому столу

Подготовка к семинару-круглому столу начинается с распределение форм участия и функции магистрантов в семинаре-круглом столе. Магистрантами осуществляется определение круга проблем и вопросов, подлежащих обсуждению; подбор основной и дополнительной литературы к теме семинара - круглого стола, а также дальнейшее изучение литературы.

Подготовка мультимедийной презентации

Презентация, согласно толковому словарю русского языка Д.Н. Ушакова: «... способ подачи информации, в котором присутствуют рисунки,

фотографии, анимация и звук». Для подготовки презентации рекомендуется использовать LibreOffice Impress (для подготовки собственно мультимедийных презентаций) и LibreOffice Writer (для составления текстового сопровождения презентации), являющихся компонентами открытого и свободного офисного пакета LibreOffice. Также допускается использование проприетарного продукта Microsoft Office (Powerpoint и Word, соответственно), однако в этом случае должны использоваться наиболее совместимые форматы .ppt, .doc (но не .pptx, .docx).

Для подготовки презентации необходимо собрать и обработать начальную информацию.

Последовательность подготовки презентации:

1. Четко сформулировать цель презентации: вы хотите свою аудиторию мотивировать, убедить, заразить какой-то идеей или просто формально отчитаться.
2. Определить каков будет формат презентации: живое выступление (тогда, сколько будет его продолжительность) или электронная рассылка (каков будет контекст презентации).
3. Отобрать всю содержательную часть для презентации и выстроить логическую цепочку представления.
4. Определить ключевые моменты в содержании текста и выделить их.
5. Определить виды визуализации (картинки) для отображения их на слайдах в соответствии с логикой, целью и спецификой материала.
6. Подобрать дизайн и форматировать слайды (количество картинок и текста, их расположение, цвет и размер).
7. Проверить визуальное восприятие презентации.

К видам визуализации относятся иллюстрации, образы, диаграммы, таблицы.

Иллюстрация – представление реально существующего зрительного ряда.

Образы – в отличие от иллюстраций – метафора. Их назначение – вызвать эмоцию и создать отношение к ней, воздействовать на аудиторию. С помощью хорошо продуманных и представляемых образов, информация может надолго остаться в памяти человека.

Диаграмма – визуализация количественных и качественных связей. Их используют для убедительной демонстрации данных, для пространственного мышления в дополнение к логическому.

Таблица – конкретный, наглядный и точный показ данных. Ее основное назначение – структурировать информацию, что порой облегчает восприятие данных аудиторией.

Практические советы по подготовке презентации.

- готовьте отдельно: печатный текст + слайды + раздаточный материал;
- слайды – визуальная подача информации, которая должна содержать
- минимум текста, максимум изображений, несущих смысловую

нагрузку, выглядеть наглядно и просто;

- текстовое содержание презентации – устная речь или чтение, которая
- должна включать аргументы, факты, доказательства и эмоции;
- рекомендуемое число слайдов 10-12;
- обязательная информация для презентации: тема, фамилия и инициалы
- выступающего; план сообщения; краткие выводы из всего сказанного; список использованных источников;
- раздаточный материал – должен обеспечивать ту же глубину и охват, что и живое выступление: люди больше доверяют тому, что они могут унести с собой, чем исчезающим изображениям, слова и слайды забываются, а раздаточный материал остается постоянным осязаемым напоминанием; раздаточный материал важно раздавать в конце презентации; раздаточный материалы должны отличаться от слайдов, должны быть более информативными.

Доклад, согласно толковому словарю русского языка Д.Н. Ушакова:

«… сообщение по заданной теме, с целью внести знания из дополнительной литературы, систематизировать материл, проиллюстрировать примерами, развивать навыки самостоятельной работы с научной литературой, познавательный интерес к научному познанию».

Тема доклада должна быть согласованна с преподавателем и соответствовать теме учебного занятия. Материалы при его подготовке, должны соответствовать научно-методическим требованиям вуза и быть указаны в докладе. Необходимо соблюдать регламент, оговоренный при получении задания. Иллюстрации должны быть достаточными, но не чрезмерными.

Работа магистранта над докладом-презентацией включает отработку умения самостоятельно обобщать материал и делать выводы в заключении, умения ориентироваться в материале и отвечать на дополнительные вопросы слушателей, отработку навыков ораторства, умения проводить диспут.

Докладчики должны знать и уметь: сообщать новую информацию; использовать технические средства; хорошо ориентироваться в теме всего семинарского занятия; дискутировать и быстро отвечать на заданные вопросы; четко выполнять установленный регламент (не более 10 минут); иметь представление о композиционной структуре доклада и др.

Структура выступления

Вступление помогает обеспечить успех выступления по любой тематике. Вступление должно содержать: название, сообщение основной идеи, современную оценку предмета изложения, краткое перечисление рассматриваемых вопросов, живую интересную форму изложения, акцентирование внимания на важных моментах, оригинальность подхода.

Основная часть, в которой выступающий должен глубоко раскрыть суть

затронутой темы, обычно строится по принципу отчета. Задача основной части – представить достаточно данных для того, чтобы слушатели заинтересовались темой и захотели ознакомиться с материалами. При этом логическая структура теоретического блока не должны даваться без наглядных пособий, аудиовизуальных и визуальных материалов.

Заключение – ясное, четкое обобщение и краткие выводы, которых всегда ждут слушатели

Написание доклада

Доклад – публичное сообщение, представляющее собой развёрнутое изложение определённой темы.

Этапы подготовки доклада:

1. Определение цели доклада.
2. Подбор необходимого материала, определяющего содержание доклада.
3. Составление плана доклада, распределение собранного материала в необходимой логической последовательности.
4. Общее знакомство с литературой и выделение среди источников главного.
5. Уточнение плана, отбор материала к каждому пункту плана.
6. Композиционное оформление доклада.
7. Заучивание, запоминание текста доклада, подготовки тезисов выступления.
8. Выступление с докладом.
9. Обсуждение доклада.
10. Оценивание доклада

Композиционное оформление доклада – это его реальная речевая внешняя структура, в ней отражается соотношение частей выступления по их цели, стилистическим особенностям, по объёму, сочетанию рациональных и эмоциональных моментов, как правило, элементами композиции доклада являются: вступление, определение предмета выступления, изложение (опровержение), заключение.

Вступление помогает обеспечить успех выступления по любой тематике.

Вступление должно содержать:

- название доклада;
- сообщение основной идеи;
- современную оценку предмета изложения;
- краткое перечисление рассматриваемых вопросов;
- интересную для слушателей форму изложения;
- акцентирование оригинальности подхода.

Выступление состоит из следующих частей:

Основная часть, в которой выступающий должен раскрыть суть темы, обычно строится по принципу отчёта. Задача основной части: представить

достаточно данных для того, чтобы слушатели заинтересовались темой и захотели ознакомиться с материалами.

Заключение - это чёткое обобщение и краткие выводы по излагаемой теме.

6. Список литературы, использованной при составлении методических рекомендаций

1. Чан У. Дж. Python. Создание приложений. - М.: Вильямс, 2013. - 816 с.
2. Horstmann, C., Cornell, G. Core Java Volume I – Fundamentals (9th Edition) / C. Horstmann, G. Cornell. – Prentice Hall, 2012. – 1008 p.
3. 5th GATE Training Course. - The University of Sheffield, 2012. - http://gate.ac.uk/wiki/TrainingCourseJune2012/
4. Friedl, J.E.F. Mastering Regular Expressions / J.E.F. Friedl. – 3rd. ed. O'Reilly Media, 2012. – 544 p.
5. Роббинс, А. Изучаем редакторы vi и Vim / А. Роббинс, Э. Хана, Л. Лэмб. – М.: Символ-Плюс, 2013. – 512 с.
6. Annotation Graph Toolkit (AGTK) – http://agtk.sourceforge.net
7. Applied Linguistics – http://www.appliedlinguistics.org
8. Center for Applied Linguistics – http://www.cal.org
9. Code::Blocks C/C++ IDE – http://www.codeblocks.org
10. Emacs – http://www.gnu.org/software/emacs
11. GATE (General Architecture for Text Engineering) – http://gate.ac.uk
12. GNU Compiler Collection – http://gcc.gnu.org
13. Helsinki Finite-State Technology – http://sourceforge.net/projects/hfst
14. Leopar Language Parser – http://leopar.loria.fr
15. NLTK (Natural Language Toolkit) – http://www.nltk.org
16. Open Natural Language Processing (OpenNLP) – http://opennlp.sourceforge.net
17. Praat – http://www.fon.hum.uva.nl/praat
18. Python Programming Language – http://www.python.org
19. Sed – http://www.gnu.org/s/sed
20. Sonic Visualizer – http://www.sonicvisualiser.org
21. Vi Improved - http://www.vim.org