Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»

Д. С. КУЗЬМЕНКОВ, Е. Ю. КУЗЬМЕНКОВА

ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ PYTHON

Практическое пособие

для студентов специальностей
1-40 01 01 «Программное обеспечение информационных технологий»,
1-40 04 01 «Информатика и технологии программирования»

Гомель
ГГУ им. Ф. Скорины
2022

УДК 004.438Python(076)
ББК 32.973.22я73
К893

Рецензенты:
кандидат технических наук К. С. Курочка,
кандидат технических наук С. Ф. Маслович

Рекомендовано к изданию научно-методическим советом
учреждения образования «Гомельский государственный
университет имени Франциска Скорины»

Кузьменков, Д. С.

К893 Основы языка программирования Python : практическое пособие / Д. С. Кузьменков, Е. Ю. Кузьменкова ; Гомельский гос. ун-т им. Ф. Скорины. – Гомель : ГГУ им. Ф. Скорины, 2022. – 47 с.

ISBN 978-985-577-885-2

Практическое пособие предназначено для оказания помощи студентам в овладении практическими приемами и навыками программирования на языке программирования Python. В нём излагается теоретический материал, даны практические задания и алгоритмы их выполнения.

Адресовано студентам 3 курса специальности 1-40 01 01 «Программное обеспечение информационных технологий», 4 курса специальности 1-40 04 01 «Информатика и технологии программирования».

УДК 004.438Python(076)
ББК 32.973.22я73

ISBN 978-985-577-885-2

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Практическое пособие призвано помочь студентам овладеть основами программирования на языке программирования Python. Язык Python в настоящее время является одним из самых популярных и востребованных в мире языков программирования. Чаще всего Python используют в веб-разработке, для него написаны различные фреймворки: Django, FastAPI, Flask, Tornado, Pyramid, CherryPy и др. Язык также широко применяется в научных исследованиях и машинном обучении. Python часто используется для автоматизации тестирования. Он нашёл свое применение и при разработке десктопных, мобильных и игровых приложений.

Издание структурно содержит две темы. В первой теме описывается синтаксис языка программирования Python: краткое описание языка, написание программ, условные операторы, операторы цикла, ключевые слова, встроенные функции, работа с числовыми данными, работа со строковыми данными, форматирование строк и др. Вторая тема посвящена специфичным типам данных языка программирования Python и работе с ними: списки, кортежи, словари, множества, функции и их аргументы, работа с декораторами, руководство по написанию кода на Python и др. В конце каждой темы есть список вопросов для самоконтроля и практические задания, направленные на закрепление приведенной в теме информации.

Практическое пособие «Основы языка программирования Python» направлено на формирование практических навыков программирования на языке Python, на закрепление полученных знаний путем выполнения индивидуальных практических заданий по темам практического пособия.

ТЕМА 1. СИНТАКСИС ЯЗЫКА ПРОГРАММИРОВАНИЯ PYTHON

1.1 Краткое описание языка программирования Python

Python – интерпретируемый объектно-ориентированный язык программирования с динамической семантикой. Язык является привлекательным для быстрой разработки приложений, так как в нем присутствуют высокоуровневые структуры данных и динамическая типизация. Python также используется в качестве сценарного языка для связи программных компонентов. В поставку Python входит обширная стандартная библиотека для решения широкого круга задач. В интернете в свободном доступе находятся библиотеки для Python по различным предметным областям: интернет-технологии, средства обработки текста, обработка изображений, механизмы доступа к базе данных (БД), искусственный интеллект, пакеты для проведения сложных вычислений и т. д.

Python имеет средства для интеграции с языками C, C++ и Java как путём встраивания (Embedding) интерпретатора в программы на этих языках, так и наоборот, через использования библиотек, написанных на этих языках, в Python программах.

Python поддерживает императивное, объектно-ориентированное и функциональное программирование. Python доступен на всех современных платформах (32, 64 bit).

Разработка языка программирования Python была начата в конце 80-х годов 20 века сотрудником голландского института Гвидо ван Россумом. Для распределённой ОС «Амёба» требовался скриптовый язык и Гвидо ван Россум начал работу над языком Python на досуге. В феврале 1991 года был опубликован исходный код языка.

3 декабря 2008 года вышла новая версия Python 3000 (Python 3.0) в этой версии устранены многие недостатки архитектуры с возможным сохранением совместимости со старыми версиями Python. На сегодня поддерживаются обе ветки развития (Python 2.x, Python 3.x). Официальный сайт для скачивания – Python.org.

Существуют 5 основных реализаций Python:

1. CPython – наиболее поддерживаемая реализация Python, написанная на C. Новые возможности языка, как правило, появляются именно в ней.

2. Jython – Python, написанный на Java. Эта реализация может быть использована как скриптовый язык для Java-приложений с применени-

ем библиотек классов Java. Jython часто применяется для написания тестов для Java-библиотек (автор Jim Hugunin).

3. Python for .net. Данная реализация основана на CPython, но является управляемым .net приложением и предоставляет доступ к библиотекам .net (автор Brian Lloyd).

4. IronPython – альтернативная реализация Python под .net. В отличие от Python for .net, это целостная реализация Python, компилирующая программы на Python непосредственно в сборке .net (автор Jim Hugunin).

5. Pypy – реализация Python, написанная на Python, даже интерпретатор байт кода написан на Python.

1.2 Написание Python программ

Программа на языке Python может состоять из одного или нескольких модулей. Каждый модуль представляет собой текстовый файл в кодировке, совместимой с 7-битной кодировкой ASCII. Для кодировок, использующих старший бит, необходимо явно указывать название кодировки. Например, модуль, комментарии или строковые литералы которого записаны в кодировке `koi8-r`, должен иметь в первой или второй строке следующую спецификацию.

```
# -*- coding:koi8-r -*-
```

Благодаря данной спецификации интерпретатор Python будет знать, как корректно переводить символы литералов Unicode-строк в Unicode. Без этой строки старые версии Python будут выдавать предупреждения на каждый модуль, в котором встречаются коды с установленным 8 битом.

Программа на Python с точки зрения интерпретатора состоит из логических строк. Одна логическая строка, как правило, располагается в одной физической. Длинные логические строки можно явно (с помощью обратной косой черты) или не явно (внутри любых скобок) разбить на несколько физических.

Пример. Логическая строка одна, а физических две.

```
a=5
print(a,"-это очень длинная строка, которая не "\
"помещается в 80 знакоместах")
```

Язык программирования Python чувствителен к регистру.

Для написания однострочного комментария используется символ #.

Вложенные инструкции объединяются в блоки по величине отступов. Отступ может быть любым, главное, чтобы в пределах одного

вложенного блока отступ был одинаков. Вложенные инструкции в Python записываются в соответствии с одним и тем же шаблоном, когда основная инструкция завершается двоеточием, вслед за которым располагается вложенный блок кода, обычно с отступом под строкой основной инструкции.

```
if a>b:
  c=a+b
  d=a*b
  print(c,d)
```

Можно расположить несколько инструкций в одной строке, разделяя их точкой с запятой.

Тело составной инструкции может располагаться в той же строке, что и тело основной, если тело составной инструкции не содержит составных инструкций.

```
if a>b: print(a)
```

1.3 Оператор условия и выбора. Циклы

Оператор `else if` – составной оператор.

```
if(a==1 and b==2 and c==3 and d==4):
    print(a,b)
else:
    print(c,d)
```

Оператор выбора – обобщение оператора `if`.

Вычислим знак числа:

```
if a<0:
   s=-1
elif a==0:#elif - сокращенный else if
   s=0
else:
   s=1
```

`elif` – это сокращенный `else if`. Без сокращения пришлось бы применять вложенный оператор `if`.

```
if a<0:
   s=-1
else:
```

```
if a==0:
 s=0
   else:
 s=1
```

Циклы

Существуют 2 вида циклов: `for` и `while`.

Пример. Удаляем первый и последний символ в строке до тех пор, пока не останется пустая строка.

```
s="Иванов меняет профессию"
while s!="":
    print(s)
    s=s[1:-1]
```

В языке Python тело цикла выделяется отступами.

Также в циклах можно использовать `break` (осуществляет выход из цикла) и `continue` (осуществляет переход к следующей итерации).

Пример. Прочитаем строки из файла и выведем только те, у которых длина больше десяти.

```
F=open("file.txt","r")
while 1:
    L=f.readline()
    If not L:
        Break
    If len(L)>10:
        print (L, end="")
       #добавляем пробел вместо перевода строки
f.close()
```

В примере организован бесконечный цикл, который прерывается только при получении из файла пустой строки (l), что и означает конец файла.

В языке Python логическое значение несет каждый объект: нули, пустые строки и последовательности, объект `none` и логический литерал `false` имеют значение ложь, а прочие – истина (`true`). Литералы `false` и `true` для обозначения логических значений появились в версии Python 2.3.

Пример. Выведем на экран «таблицу умножения».

```
for i in range(1,10): #[1,10)
    for j in range (1,10):
        print ("%2i"%(i*j),end=" ")
    print() # перевод строки
```

В примере цикл `for` является вложенным. Функция `range` порождает список целых чисел из полуоткрытого интервала `[1,10)`. Полуоткрытые интервалы общеприняты в Python. Функция `range(len(s))` порождает список индексов для списка `s` (в Python-последовательности первый элемент имеет индекс 0).

Для красивого вывода таблицы умножения применяется операция форматирования `%`. Строка форматирования, которая задаётся слева, строится почти по тем же правилам, что и строка форматирования для функции printf в языке C.

1.4 Ключевые слова. Встроенные функции

Ключевые слова

`false` – ложь;

`true` – правда;

`none` – «пустой» объект;

`and` – логическое *и*;

`with / as` – менеджер контекста;

`assert условие` – возбуждает исключение, если условие ложно;

`break` – выход из цикла;

`class` – пользовательский тип, состоящий из методов и атрибутов;

`continue` – переход на следующую итерацию цикла;

`def` – определение функции;

`del` – удаление объекта;

`elif` – в противном случае, если;

`else` – иначе;

`except` – перехватить исключение;

`finally` – вкупе с инструкцией `try` выполняет инструкции независимо от того, было ли исключение или нет;

`for` – цикл `for`;

`from` – импорт нескольких функций из модуля;

`global` – позволяет сделать значение переменной, присвоенное ей внутри функции, доступным и за пределами этой функции;

`if` – если;

`import` – импорт модуля;

`in` – проверка на вхождение;

`is` – ссылаются ли 2 объекта на одно и то же место в памяти;

`lambda` – определение анонимной функции;

`nonlocal` – позволяет сделать значение переменной, присвоенное ей внутри функции, доступным в объемлющей инструкции;

`not` – логическое *не*;

`or` – логическое *или*;
`pass` – ничего не делающая конструкция;
`raise` – возбудить исключение;
`return` – вернуть результат;
`try` – выполнить инструкции, перехватывая исключения;
`while` – цикл `while`;
`yield` – определение функции-генератора.

Встроенные функции, выполняющие преобразование типов

`bool(x)` – преобразование к типу `bool`, использует стандартную процедуру проверки истинности. Если `x` является ложным или опущен, возвращает значение `false`, в противном случае она возвращает `true`;

`bytearray([источник [, кодировка [ошибки]]])` – преобразование к `bytearray`. `bytearray` – изменяемая последовательность целых чисел в диапазоне $0 \leq X < 256$. Функция, вызванная без аргументов, возвращает пустой массив байт;

`bytes([источник [,кодировка [ошибки]]])` – возвращает объект типа `bytes`, который является неизменяемой последовательностью целых чисел в диапазоне $0 \leq X < 256$. Аргументы конструктора интерпретируются как для `bytearray()`;

`complex([real[, imag]])` – преобразование к комплексному числу;

`dict([object])` – преобразование к словарю;

`float([X])` – преобразование к числу с плавающей точкой. Если аргумент не указан, возвращается 0.0;

`frozenset([последовательность])` – возвращает неизменяемое множество;

`int([object], [основание системы счисления])` – преобразование к целому числу;

`list([object])` – создает список;

`memoryview([object])` – создает объект `memoryview`;

`object()` – возвращает безликий объект, являющийся базовым для всех объектов;

`range([start=0], stop, [step=1])` – арифметическая прогрессия от `start` до `stop` с шагом `step`;

`set([object])` – создает множество;

`slice([start=0], stop, [step=1])` – объект среза от `start` до `stop` с шагом `step`;

`str([object], [кодировка], [ошибки])` – строковое представление объекта. Использует метод `__str__`;

`tuple(obj)` – преобразование к кортежу.

Основные встроенные функции

`abs(x)` – возвращает абсолютную величину (модуль числа);

`all(последовательность)` – возвращает `true`, если все элементы истинные (или если последовательность пуста);

`any(последовательность)` – возвращает `true`, если хотя бы один элемент – истина. Для пустой последовательности возвращает `false`;

`bin(x)` – преобразование целого числа в двоичную строку;

`chr(x)` – возвращает односимвольную строку, код символа которой равен `x`;

`dir([object])` – список имен объекта, а если объект не указан, список имен в текущей локальной области видимости;

`divmod(a, b)` – возвращает частное и остаток от деления `a` на `b`;

`exec(object[, globals[, locals]])` – выполняет программный код на Python;

`format(value[,format_spec])` – форматирование (обычно форматирование строки);

`help([object])` – вызов встроенной справочной системы;

`hex(x)` – преобразование целого числа в шестнадцатеричную строку;

`input([prompt])` – возвращает введенную пользователем строку. `Prompt` – подсказка пользователю;

`len(x)` – возвращает число элементов в указанном объекте;

`max(iter, [args ...] * [, key])` – максимальный элемент последовательности;

`min(iter, [args ...] * [, key])` – минимальный элемент последовательности;

`oct(x)` – преобразование целого числа в восьмеричную строку;

`open(file, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True)` – открывает файл и возвращает соответствующий поток;

`ord(c)` – код символа;

`pow(x, y[, r])` – `( x ** y ) % r` (`x` в степени `y` по модулю `r`).

`print([object, ...], *, sep=" ", end='\n', file=sys.stdout)` – печать;

`round(X [, N])` – округление до `N` знаков после запятой;

`sum(iter, start=0)` – сумма членов последовательности;

`type(object)` – возвращает тип объекта.

1.5 Основные различия Python 2 и Python 3

1. Функция `print`. В 3-й версии оператор `print` был заменён функцией:

```
print ”ответ”,2*2          print x,
print (”ответ”,2*2)        print(x,end=” ”)
```

2. Операторы сравнения. Они создают исключение `Type Error`, когда операнды не упорядочиваемы (`none<none` в Python 2 возвращает `false`, а в Python 3 – ошибку `Type Error`). Следовательно, сортировка списка с разными типами данных больше не имеет смысла, так как все элементы должны быть сравнимы друг с другом.

3. Целые числа. Тип `long` переименован в `int`. Выражение вида ½ возвращает `false`, `1//2` – отсекает дробную часть. Константа `sys.maxint` была удалена, т. к. уже не существует предела значений целых чисел.

4. Текст, Unicode и 8-битные строки. Python 3 использует понятия текста и данных вместо Unicode-строк и 8-битных строк. Весь текст – Unicode. Типом, используемым для хранения текста, является `str`, а тип, используемый для хранения данных, – `bytes`. В Python 3 любая попытка скомбинировать текст и данные вызывает `Type Error`. Кодировка исходного кода по умолчанию UTF-8, символы, не входящие в кодировку ASCII, разрешены в идентификаторах.

5. Был изменен синтаксис некоторых операторов и встроенных функций. Например: оператор `file` был удален, вместо него добавлена функция `open()`.

1.6 Работа с числами

Целые числа в Python 3 относятся к типу `int`. Над ними можно выполнять набор обычных математических операций:

`x + y` – сложение;
`x - y` – вычитание;
`x * y` – умножение;
`x / y` – деление;
`x // y` – получение целой части от деления;
`x % y` – остаток от деления;
`-x` – смена знака числа;
`x ** y` – возведение в степень.

Целые числа в Python 3 поддерживают длинную арифметику (например, 3**150).

Над целыми числами также можно производить битовые операции:

`x | y` – побитовое *или*;

`x ^ y` – побитовое *исключающее или*;

`x & y` – побитовое *и*;

`x << n` – битовый сдвиг влево;

`x >> y` – битовый сдвиг вправо;

`~x` – инверсия битов.

В Python 3 можно работать с числами в двоичной, восьмеричной, десятичной и шестнадцатеричной системах счисления.

Вещественные числа относятся к типу `float`. Данные типа `float` являются неточными, для более высокой точности используют типы `decimal` и `fraction`. Также вещественные числа не поддерживают длинную арифметику.

Модуль `math` содержит сложные математические функции.

```
import math
math.pi      #Результат 3.141592653589793
math.sqrt(9)     #Результат 3
```

Модуль `random` содержит генераторы случайных чисел и функции случайного выбора.

```
import random
random.random()      #Результат 0.1516
```

В Python также встроены комплексные числа.

```
x=complex(1,2)
print(x)     #Результат (1+2j)
```

Для работы с комплексными числами используется модуль `cmath`.

1.7 Работа со строками

Строки можно заключать в кавычки и апострофы. Это сделано для того, чтобы можно было вставлять в строковые литералы символы кавычек или апострофов без использования экранирования.

```
S=”Питон”3’
S=”Питон’3”
```

Экранированные последовательности – служебные символы, которые позволяют ставить символы, сложно вводимые с клавиатуры:

`\n` – перевод строки;

`\f` – перевод страницы;

`\t` – горизонтальная табуляция;

`\v` – вертикальная табуляция.

Если перед открывающей кавычкой стоит символ `r`, причем в любом регистре, то механизм экранирования отключается. Но строка, начинающаяся с `r` (сырая строка) не может заканчиваться обратным слешем (\). Для этого можно:

```
S=r'\n\n\\'[:-1]
S=r'\n\n'+'\\'
```

Строки, заключенные в тройные кавычки (или апострофы), можно использовать для записи многострочных блоков текста (комментариев).

```
c='''это будет большой
          …трехстрочный
          …блок текста'''
print(c)
```

Базовые операции над строками

1. Конкатенация.

```
s1='A'
s2='B'
print(s1+s2)
#'AB'
```

2. Дублирование строки.

```
print(s1*3)
#'AAA'
```

3. Доступ по индексу.

```
s='Python'
s[0]
#'P'
s[2]
#'t'
s[-3]
#'h'
```

4. Длина строки.

```
len(s)   #6
```

5. Извлечение среза.

Оператор извлечения среза – `[X:Y]`, где `X` – начало среза, `Y` – окончание. Символ с номером `Y` в срез не входит. По умолчанию первый индекс равен 0, а второй – длине строки.

```
s=Python
s[2:5]        #'tho'
s[2:-2]        #'th'
s[:5]         #'Pytho'
s[1:]         #'ython'
s[:]           #'Python'
```

Можно также задать шаг, с которым нужно извлекать срез.

```
s[::-1]    #'nohtyP'
s[2::2]    #'to'
s[2:4:-1] #''
s[4:2:-1] #'oh'
```

При вызове методов необходимо помнить, что строки в Python относятся к категории *неизменяемых последовательностей*, то есть все функции и методы могут лишь создавать новую строку. Следовательно, все строковые методы возвращают новую строку, которую далее можно присвоить переменной.

```
s='Python'
s[1]= 'i'
#Type Erorr
s=s[0]+'i'+s[2:]
#'Pithon'
```

Некоторые функции и методы обработки строк

`s.find(str,[start,][end])` – поиск подстроки в строке, возвращает номер первого вхождения или -1;

`s.rfind(str,[start,][end])` – поиск подстроки в строке, возвращает номер последнего вхождения или -1;

`s.split(символ)` – разбиение строки по разделителю;

`s.isdigit()` – проверяет, состоит ли строка из цифр;

`s.isalpha()` – проверяет, состоит ли строка из букв;

`s.islower()` – проверяет, состоит ли строка из символов в нижнем регистре;

`s.isupper()` – проверяет, состоит ли строка из символов в верхнем регистре;

`s.istitle()` – проверяет, начинаются ли слова в строке с заглавной буквы;

`s.upper()` – преобразование строки к верхнему регистру;

`s.lower()` – преобразование строки к нижнему регистру;

`ord(символ)` – перевод символа в его код ASCII;

`chr(число)` – перевод кода ASCII в символ;

`s.istrip([chars])` – удаление пробельных символов в начале строки;

`s.rstrip([chars])` – удаление пробельных символов в конце строки;

`s.strip([chars])` – удаление пробельных символов в начале и в конце строки;

`s.swapcase()` – переводит символы нижнего регистра в верхний, а верхнего – в нижний;

`s.title()` – первую букву каждого слова переводит в верхний регистр, а верхнего – в нижний;

`s.format(*args,**kwards)` – форматирование строки.

1.8 Форматирование строк. Метод format

Часто возникают ситуации, когда необходимо составить строку, подставив в нее некоторые данные, полученные в процессе выполнения программы (пользовательский ввод, чтение данных из файлов). Подстановку данных можно сделать с помощью форматирования строк, которая реализуется оператором `%` или методом `format`.

Форматирование с помощью метода format

Если для подстановки требуется только один аргумент, то значение – сам аргумент.

```
s='Olga'
'Hello, {}!'.format(s)
#'Hello, Olga!'
```

Если требуется несколько аргументов, то значениями будут являться все аргументы со строками подстановки.

```
'{0},{1},{2}'.format('a','b','c')
#'a,b,c'
'{},{},{}'.format('a','b','c')
#'a,b,c'
'{2},{1},{0}'.format('a','b','c')
#'c,b,a'
'{0},{1},{0}'.format('abra','cad')
#'abracadabra'
coord={'x':'5','y':'-2'}
'координаты:{x},{y}'.format(**coord)
'координаты:5,-2'
```

Синтаксис оператора format

```
поле замены::"{"[имя поля]["!"преобразование]
[":"спецификация"}"
имя поля =arg_name("."имя атрибута|"["индекс"]")
преобразовние="r"(внутр.представление)|
"s"(чел.представление)
спецификация=[[fill]align][sign][#][0][width][,]
[.precision][type]
```

Спецификация оператора format

`fill` – символ-заполнитель – любой символ, кроме `{ }`;

`align` – выравнивание проводится с помощью символа заполнителя. Возможные значения:

< – символы заполнители будут справа (выравнивание по левому краю); задано по умолчанию;

> – выравнивание объекта по правому краю;

= – заполнитель будет после знака, но перед цифрами;

^ – выравнивание по центру.

Операнд `sign` – знак, который используется только для чисел. Возможные значения:

+ – знак должен быть использован для всех чисел;

– – минус для отрицательных чисел, ничего для положительных;

пробел – минус для отрицательных чисел, пробел для положительных.

Поле `type` может принимать следующие значения:

`d, i, u` – десятичное число;

`o` – число в 8-ричной системе счисления;

`x` – число в 16-ричной системе счисления (буквы в нижнем регистре), `X` (буква в верхнем регистре);

e – число с плавающей точкой с экспонентой (экспонента в нижнем регистре) E – экспонента в верхнем регистре;

f, F – число с плавающей точкой;

g – число с плавающей точкой с экспонентой (экспонента в нижнем регистре), если она меньше чем –4 или меньше точности, иначе – обычный формат. G – экспонента в верхнем регистре;

c – символ (строка из одного символа или число – код символа);

s – строка;

r – литерал, заключается при выводе в апострофы;

% – число умножается на 100, отображается с плавающей точкой, а за ним знак процента;

b – число в двоичной системе счисления;

– задает префиксы систем счисления.

```
"{:b}".format(0b11) #'11'
"{:#b}".format(0b11) #'0b11'
"{:,}".format(1234567890)
#'1,234,567,890'
```

Заполнение нулями:

```
"{:04d}".format(9)
#'0009'
```

Выравнивание по левому краю:

```
"{:<30}".format('по левому краю')
```

Выравнивание по центру с заполнением звездочками:

```
"{:*^30}".format('по центру')
```

Пример с координатами:

```
coord=(3,5)
'x:{0[0]};y:{0[1]}'.format(coord)
#'x:5;y:5'
```

Пример с процентами:

```
a=5
b=7
'5 на 7 в %:{:.2%}'.format(a/b)
#'5 на 7 в %:71,43%'
```

Практическое задание

На языке Python написать программу Python, реализующую решение задачи, с использованием строкового типа данных. Программу просчитать для различных исходных данных.

Вариант 1.

Дан текст, содержащий не более 250 символов. В тех словах, которые заканчиваются сочетанием букв ING, заменить это окончание на ED.

Вариант 2.

Выбрать в тексте первое по порядку слово с наибольшим числом вхождений в него буквы *и*.

Вариант 3.

Дан текст. Слова в тексте разделены пробелами. Вычеркнуть из текста все повторяющиеся слова.

Вариант 4.

Дан текст. Все слова, длина которых больше в 2 раза длины слова минимальной длины, заменить на слово минимальной длины.

Вариант 5.

Дан текст. Получить все символы, отличные от букв и цифр.

Вариант 6.

Дан текст. Создать новый текст, в котором все слова нечетной длины заменены на слово минимальной длины.

Вариант 7.

Определить, сколько раз в тексте встречаются слова минимальной длины.

Вариант 8.

Дан текст, содержащий не более 250 символов. В тех словах, которые заканчиваются сочетанием букв «ый», заменить это сочетание на «ая».

Вариант 9.

Дан текст. Создать новый текст, в котором все слова, которые начинаются на ту же букву, что и слово минимальной длины, продублировать.

Вариант 10.

Определить, сколько раз в тексте встречаются слова максимальной длины.

Вариант 11.

Дан текст. Создать новый текст путем вычеркивания из исходного текста: а) слов четной длины, если максимальное слово имеет четную длину; б) слов нечетной длины, если максимальное слово имеет нечетную длину.

Вариант 12.

Создать новый текст, содержащий все слова исходного текста, которые оканчиваются на ту же букву, что и слово максимальной длины.

Вариант 13.

Дан текст. Создать новый текст, в котором все слова нечетной длины заменены на слово максимальной длины.

Вариант 14.

В тексте вставить вместо одного пробела запятую и пробел, вместо двух пробелов – двоеточие и пробел, вместо трех и более пробелов – тире и пробел.

Вариант 15.

Создать новый текст, содержащий все слова исходного текста, которые оканчиваются на ту же букву, что и слово максимальной длины. Из исходного текста эти слова удалить.

Вопросы для самоконтроля

1. В каком году был опубликован исходный код языка программирования Python?
2. Кто автор языка программирования Python?
3. В каком году появилась версия языка Python 3?
4. Перечислите основные сферы использования языка программирования Python.
5. Из чего с точки зрения интерпретатора состоит программа на Python?
6. Какой символ используется для написания комментария в строке?
7. Какие существуют операторы циклов в Python?
8. Какие ключевые слова языка Python используются для обработки исключений?
9. Перечислите основные различия синтаксиса Python 2 и Python 3.
10. В каких системах счисления можно работать с числами в Python?
11. Какие символы используются для описания строковых литералов в Python?
12. Что такое экранированные последовательности?
13. Что необходимо помнить при работе со строковыми литералами в Python?
14. Для чего в Python используется метод format?
15. Перечислите основные спецификации оператора format.

ТЕМА 2. СПЕЦИФИЧНЫЕ ТИПЫ ДАННЫХ ЯЗЫКА PYTHON И РАБОТА С НИМИ

2.1 Списки (list)

Списки в Python – упорядоченные изменяемые коллекции произвольных типов (почти как массивы, но типы могут отличаться).

Создать списки можно несколькими способами:

1. Встроенная функция `list`.

```
list('список')
#['с', 'п', 'и', 'с', 'о', 'к']
```

2. При помощи литерала.

```
s=[]#пустой список
l=['с', 'п',[ 'исок'],2]
#['с', 'п ',[ 'исок'],2]
```

Список может содержать любое количество любых объектов (в том числе и сложенные списки) или не содержать ничего.

3. Использование генераторов списков – способ построить новый список, применяя выражение к каждому элементу последовательности. Генераторы списка очень похожи на цикл `for`.

```
[c*3 for c in 'list']
c
#['lll','iii','sss','ttt']
[c*3 for c in 'list' if c!='i']
c
#['lll','sss','ttt']
```

Нумерация в списках начинается с нуля.

Основные функции и методы обработки списков

`list.append(x)` – добавляет элемент в конец списка;

`list.extend(L)` – расширяет список `list`, добавляя в конец все элементы списка `L`;

`list.insert(i,x)` – вставляет на `i`-ю позицию значение `x`;

`list.remove(x)` – удаляет первый элемент в списке, имеющий значение `x`;

`list.pop([i])` – удаляет из списка `i`-й элемент и возвращает его. Если индекс не указан, то удаляется последний элемент;

`list.index(x,[start[,end]])` – возвращает положение первого элемента от start до end со значением `x`;

`list.count(x)` – возвращает количество элементов со значением `x`;

`list.sort([key=func])` – сортирует список на основе функции;

`list.reverse()` – переворачивает список;

`list.copy()` – поверхностная копия списка (появилась в Python 3.3);

`list.clear()` – очищает список (появилась в Python 3.3).

Методы списков изменяют сам список, поэтому результат выполнения нельзя записывать в эту переменную.

```
l=[1,4,7,3,2]
l.sort()
l
#[1,2,3,4,7]
l=l.sort()
print(l)
#None
a=[1,2016,2,2016,29]
print(a.count(2016),a.count(1))
#2   1
a.insert(2,31)
a.append(3)
a
#[1,2016,31,2,2016,29,3]
a.index(2016)
#1
a.remove(2016)
#[1,31,2,2016,29,3]
```

В некоторых случаях для увеличения производительности или для удобства работы списки заменяют менее гибкими массивами, которые содержаться в модуле `array`. Массивы, как и в большинстве других языков программирования, имеют ограничения на тип данных и размер каждого элемента.

2.2 Индексы и срезы

Нумерация элементов в Python начинается с нуля. При попытке доступа к несуществующему индексу возникает ошибка `IndexError`.

Взять элемент по индексу можно у списков, строк и кортежей. В Python поддерживаются отрицательные индексы, при этом нумерация идет с конца.

```
a=[1,7,10,2]
a[0]     #1
a[4]
# IndexError:list index out of range
>>>a[-1]     #2
```

Срезы

`item[start:stop:step]` берёт срез от номера `start` до `stop` (не включая его) с шагом `step`. По умолчанию: `start = 0`, `stop=` длине объекта, `step = 1`. Какие-либо параметры могут быть опущены. Все эти параметры также могут быть отрицательными.

```
a=[1,7,10,2]
a[:]
#[1,7,10,2]
a[1:]
#[7,10,2]
a[:3]
#[1,7,10]
a[::2]
#[1,10]
a[::-1]
#[2,10,7,1]
a[:-2]
#[1,7]
a[-2::-1]
#[10,7,1]
a[1:3:-1]
#[]
```

В последнем примере возвращается пустой список, т. к. `start<stop`, а `step` – отрицательный.

С помощью срезов можно не только извлекать элементы, но и добавлять и удалять элементы.

```
a=[1,7,10,2]
a[1:3]=[0,0,0]
#[1, 0, 0, 0, 2]
del a[:-3]
#[0,0,2]
```

```
a[1:3]=[0,0,0]
#[0,0,0,0]
```

2.3 Кортежи (tuple)

Кортежи – неизменяемые списки. Зачем нужны?

1. Защита от дурака, т. е. защита от случайных и намеренных изменений.
2. Меньший размер по сравнению со списками.
3. Возможность использования кортежей в качестве ключей словаря.

Работа с кортежами похожа на работу со списками.

Кортежи можно задать двумя способами:

1. С помощью встроенной функции `tuple()`.

```
a=tuple() #создаем пустой кортеж
```

2. С помощью литерала кортежа.

```
a=('s')
print(a)
#'s'
```

В результате получилась строка. Описывать кортеж в данном случае необходимо, используя запятую.

```
a=('s',)
a
#('s',)
```

Сами по себе круглые скобки ничего не значат, точнее они означают, что внутри них находится одна инструкция, которая может быть отделена пробелами, переносом строк и т. д. Кортеж можно описать и без использования круглых скобок.

```
a='s'
a
#('s',)
A=tuple('нэт Галинки')
>>>a
#('н','э','т',' ','Г','а','л','и','н','к','и')
```

Над кортежами допустимы все операции над списками, не изменяющие список (сложение, умножение на число, методы `index()` и `count()` и т. д.).

Использовать синтаксис кортежей можно и в левой части оператора присваивания. В этом случае на основе вычисленных справа значений формируется кортеж и связывается один в один с именами в левой части.

Пример. Поменяем местами значения двух переменных

```
a,b=b,a
```

2.4 Словари (dict)

Словари в Python – неупорядоченные коллекции произвольных объектов с доступом по ключу (их иногда называют ассоциативными массивами, или хеш-таблицами).

Словарь можно создать несколькими способами:

1. С помощью литерала.

```
d={}     #пустой словарь
d={'phyth':1,'math':2}
```

2. С помощью функции `dict`.

```
d=dict(phyth='ASOI',math='VMP, MPU')
d=dict([(1,1),(2,4)])
d={1:1,2:4}
```

3. С помощью метода `fromkeys`.

```
d=dict.fromkeys(['a','b'])
d
#{'b':None,'a':None}
d=dict.fromkeys(['a','b'],100)
d
#{'b':100,'a':100}
```

4. С помощью генераторов словарей.

```
d={a:a**2 for a in range(7)}
d
#{0:0,1:1,2:4,3:9,4:16,5:25,6:36}
```

Добавим записи в словарь и извлечем из него значения ключей.

```
d={1:2,2:4,3:9}
d[1]
#2
d[4]=4**2
d
```

```
#{1:2,2:4,3:9,4:16}
>>>d['1']
# Error…
```

Присвоение новому ключу расширяет словарь, присвоение по существующему ключу перезаписывает его, попытка извлечения несуществующего ключа вызовет ошибку.

Методы словарей

`dict.clear()` – очищает словарь;

`dict.copy()` – возвращает копию словаря;

`dict.get(key[,default])` – возвращает значение ключа, но если его нет, возвращает `default` (по умолчанию `none`);

`dict.items()` – возвращает пары (ключ, значение);

`dict.keys()`– возвращает ключи в словаре;

`dict.pop(key[,default])` – удаляет ключ и возвращает его значение. Если ключа нет, то возвращает `default` (по умолчанию будет ошибка);

`dict.popitem()` – удаляет и возвращает пару (ключ, значение). Если словарь пустой – исключение `KeyError`;

`dict.setdefault(key[,default])` – возвращает значение ключа, но если его нет, то не создает исключение, а создает ключ со значением `default` (по умолчанию `none`);

`dict.update([other])` – обновляет словарь, добавляя пары (ключ, значение) из `other`. Существующие ключи перезаписываются;

`dict.values()` – возвращает значения в словаре.

2.5 Множества (set и frosenset)

Множество в Python – контейнер, содержащий не повторяющиеся элементы в случайном порядке.

Создадим множество различными способами.

```
a=set()
a
set()
a=set('GALINKA')
a
#{'N', 'A', 'L', 'G', 'K', 'I'}
a={'a','b','c','d'}
a
#{'b','c','a','d'}
```

```
a={i**2 for i in range(10)}#
a
#{0,1,4,81,64,9,16,49,25,36}
a={}   # так низзя
 type(a)
#<class 'dict'>
```

Маленькие и большие буквы во множествах различаются.

Используем множества для удаления повторяющихся элементов.

```
words=['Galinka','Irinka','Alex', 'Galinka']
set(words)
#{'Galinka','Irinka','Alex'}
```

Операции над множествами

`len()` – число элементов во множестве;

`x in (s)` – принадлежит ли `x` множеству `s`;

`set.isdisjoint(other)` – истина, если множества `set` и `other` не имеют общих элементов;

`set==other` – равенство множеств;

`set.issubset(other)` или `set<=other` – все элементы `set` принадлежат `other`;

`set.issuperset(other)` или `set>=other` – все элементы `other` принадлежат `set`;

`set.copy` – копия множества;

`set.union(other,…)` или `set|other|…` – объединение нескольких множеств;

`set.intersection(other,…)` или `set&other&…` – пересечение множеств;

`set.difference(other,…)` или `set-other-…` – разность множеств (множество из всех элементов `set`, не принадлежащее не одному из `other`);

`set.symmetric_difference(other)` или `set^other` – множество из элементов, встречающихся в одном множестве, но не встречающихся в обоих.

Операции, изменяющие множества

Для последних четырех операций над множествами существуют функции, выполняющие аналогичные действия над множествами и заканчивающиеся на `update`:

`set.add(elem)` – добавляет элемент во множество;

`set.remove(elem)` – удаляет элемент из множества. Если такого элемента не существует, то возникает `KeyError`;

`set.discard(elem)` – удаляет элемент, если он находится во множестве;

`set.pop()` – удаляет первый элемент из множества;

`set.clear()` – очищает множества.

Единственное отличие `set` от `frozenset` заключается в том, что `set` – изменяемый тип данных, а `frozenset` – нет.

```
a=set('Galinka')
b=frozenset('Galinka')
a==b
#true
type(a-b)
#<class 'set'>
a.add('!')
# Galinka!
b.add('!')
# Error 'frozenset' object has no attribute 'add'
```

2.6 Функции и их аргументы

Обычно функция в Python определяется с помощью инструкции `def`.

```
Def add(x,y)
    Return x+y
```

Инструкция `Return` говорит о том, что необходимо вернуть значение суммы.

Вызов функции.

```
add(1,10)
//11
add('ab','cd')
//'abcd'
```

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи и даже функции).

Пример.

```
def newfunc(n):
```

```
    def myfunc(x):
        return x+n
    return myfunc
…
new=newfunc(100)
new(200)
//выведет 300
```

Если функция не заканчивается инструкцией `return`, то она возвращает `None`.

```
def func():
   pass
print(func())
//None
```

В Python также распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.

```
def func(a,b,c=2):#c- необязат.арг.
    Return a+b+c
func(1,2)
//5
func(1,2,3)
//6
func(a=1,b=3)
//6
func(a=3,c=6)
//TypeError: func () takes at least 2 arguments (2 given)
```

Функция также может принимать переменное количество позиционных аргументов (перед именем аргумента ставится *).

```
Def func(*arg)
    Return args
…
func(1,2,3,'abc')
//(1,2,3,'abc')
func()
//()
func(1)
//(1,)
```

`args` – кортеж из всех переданных аргументов функции; с переменной `args` можно работать так же, как и с кортежем.

Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится **.

```
def func(**args2):
    Return args2
func(a=1,b=2,c=3)
//{'b':2,'a':1,'c':3}
func()
//{}
func(a='Python')
//{'a':'Python'}
```

В переменной `args2` мы храним словарь, к которому можно применять соответствующие методы.

Анонимные функции. Инструкция lambda

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции `lambda`, их не обязательно присваивать переменной.

```
func=lambda x,y:x+y
func(1,2)
//3
>func('a','b')
//'ab'
(lambda x,y:x+y)(1,2)
//3
(lambda x,y:x+y('a','b'))
//'ab'
```

`lambda`-функции не требуется инструкция `return`.

```
func=lambda *args:args
func(1,2,3,4)
//(1,2,3,4)
```

Если в теле функции написать `global` и указать имена переменных, то одноименные глобальные переменные будут меняться при изменении одноименных локальных переменных.

2.7 Функция input

Когда выполняется функция `input()`, то поток выполнения программы останавливается в ожидании данных, которые пользователь должен ввести с клавиатуры. После ввода данных и нажатия Enter функция завершает свое выполнение и возвращает результат, который представляет собой ***строку символов***, введенных пользователем. Чтобы указать пользователю, что необходимо вводить, используется необязательный строковый аргумент `prompt`.

```
Input('Введите размерность вектора!')
```

Данные возвращаются в виде строки.

В более ранних версиях Python были две встроенные функции, позволяющие получать данные с клавиатуры: `raw-input()`, возвращающая в программу строку, и `input()`, возвращающая в программу число. Начиная с версии Python 3.0, если требуется получить число, то результат функции `input()` изменяют с помощью функций `int()` или `float()`.

```
n=int(input('введите размерность вектора!'))
```

2.8 Декораторы

Функции в Python являются объектами, т. е. их можно возвращать из другой функции или передавать в качестве аргумента. Функция в Python может быть определена и внутри другой функции. *Декоратор* – это «обёртка», позволяющая изменить поведение функции, не изменяя её код.

```
def my_decorator(f_to_decorate):
# Внутри себя декоратор определяет функцию-"обёртку".
#Она будет обёрнута вокруг декорируемой, получая
#возможность исполнять произвольный код до и после неё.
    def obertka():
        print("Код,который отработает до вызова функции")
        f_to_decorate() # Сама функция
        print("Код, срабатывающий после вызова функции")
        # Вернём эту функцию
    return obertka
# код нижеследующей функции нельзя изменять(очень нужно)
```

```
def alone_function():
    print("Я - одинокая 30-летняя функция, меня уже не
    изменить!")
…
alone_function()
Я - одинокая 30-летняя функция, меня уже не изменить!
# чтобы изменить поведение функции, можно декорировать
# её, т.е просто передать декоратору, который обернет
#исходную функцию в любой код, который нам необходим,
#и вернёт новую, готовую к использованию функцию:
alone_function_decorated = my_decorator(alone_function)
alone_function_decorated()
Код, который отработает до вызова функции
Я одинокая 30-летняя функция, меня уже не изменить!
Код, срабатывающий после вызова функции
```

Чтобы каждый раз время вызова `alone_function()` вместо неё вызывалась `alone_function_decorated()`, необходимо перезаписать

```
alone_function().
alone_function = my_decorator(alone_function)
alone_function()
Код, который отработает до вызова функции
Я одинокая 30-летняя функция, меня уже не изменить!
Код, срабатывающий после вызова функции
```

Предыдущий пример можно записать и по-другому:

```
@my_decorator
def alone_function2():
    print("Не трогайте меня")
…
alone_function2()
Код, который отработает до вызова функции
Не трогайте меня!
Код, срабатывающий после вызова функции
alone_function2 = my_decorator(alone_function2)
```

Можно использовать несколько декораторов для одной функции.

```
def bread(func):
    def wrapper():
        print()
        func()
```

```
        print("<\_____/>")
    return wrapper
def ingredients(func):
    def wrapper():
        print("#помидоры#")
        func()
        print("~салат~")
    return wrapper
def sandwich(food="--ветчина--"):
    print(food)
sandwich()
sandwich = bread(ingredients(sandwich))
sandwich()
```

```
--ветчина--

#помидоры#
--ветчина--
~салат~
<\_____/>
```

Можно использовать синтаксис декораторов, т. е. вместо последних четырёх строк запишем:

```
@bread
    @ingredients
    def sandwich(food="--ветчина--"):
        print(food)
sandwich()
```

```
#помидоры#
--ветчина--
~салат~
<\_____/>
```

Важен порядок декорирования.

```
sandwich = ingredients(bread(sandwich))
sandwich()
```

```
#помидоры#

--ветчина--
<\_____/>
~салат~
```

Передача декоратором аргументов в функцию

```
def a_decorator_passing_arguments(f_to_decorate):
    def a_wrapper_accepting_arguments(arg1, arg2):
        print("Смотри, что я получил:", arg1, arg2)
        f_to_decorate(arg1, arg2)
    return a_wrapper_accepting_arguments
# Теперь, при вызове функции, которая возвращает
#декоратор, вызываем её "обёртку", передаём ей аргумен-
ты
#и она в свою очередь передаёт их декорируемой функции
@a_decorator_passing_arguments
def print_full_name(first_name, last_name):
    print("Тебя зовут", first_name, last_name)
print_full_name("Конь","в пальто")
```

Декорирование методов

Функции и методы в Python – это практически одно и то же, за исключением того, что методы всегда ожидают в качестве первого аргумента ссылку на сам объект (`self`).

```
def decor1(m_to_decorate):
    def wrapper(self, lie):
        lie -= 2
        return m_to_decorate(self, lie)
    return wrapper
class Ksusha:
    def __init__(self):
        self.age = 19
    @decor1
    def sayYourAge(self, lie):
        print("Мне {} лет, а ты бы сколько
        дал?".format(self.age + lie))
VZ= Ksusha ()
VZ.sayYourAge(-2)
```

Создадим максимально общий декоратор, чтобы его можно было применить к любой функции или методу.

```
def a_decorator_passing_arbitrary_args(f_to_decorate):
    # Данная "обёртка" принимает любые аргументы
    def a_wrapper_accepting_arbitrary_args(*args,
    **kwargs):
        print("Передали ли мне что-нибудь?:")
        print(args)
```

```
        print(kwargs)
        f_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_args
@a_decorator_passing_arbitrary_args
def f_with_named_args(a, b, c, d="Почему нет?"):
    print("Любят ли {}, {} и {} ООП?
{}".format(a,b,c,d))
f_with_named_args("Иринка", "Александра", "Галинка",
d="Определенно!")
Передали ли мне что-нибудь?:
('Иринка', 'Александра', ' Галинка')
{'d': 'Определенно!'}
Любят ли Иринка, Александра и Галинка ООП? Определенно!
class Irinka(object):
    def __init__(self):
        self.age = 20
    @a_decorator_passing_arbitrary_args
    def sayYourAge(self, lie=-2):
# Теперь мы можем указать значение по умолчанию
        print("Мне {} лет, а ты бы сколько
        дал?".format(self.age + lie))
i = Irinka()
i.sayYourAge()
Передали ли мне что-нибудь?:
(<__main__.Irinka object at 0x7f6373017780>,)
{}
Мне 18 лет, а ты бы сколько дал?
```

Напишем декоратор, принимающий аргументы.

```
def decorator_maker():
    print("Я создаю декораторы! Я буду вызван только раз:
    когда ты попросишь меня создать декоратор.")
    def my_decorator(func):
        print("Я - декоратор! Я буду вызван только раз: в
        момент декорирования функции.")
        def wrapped():
            print ("Я - обёртка вокруг декорируемой
            функции.\n"
            "Я буду вызвана каждый раз, когда ты
            вызываешь декорируемую функцию.\n"
            "Я возвращаю результат работы декорируемой
            функции.")
```

```
            return func()
        print("Я возвращаю обёрнутую функцию.")
        return wrapped
    print("Я возвращаю декоратор.")
    return my_decorator
#Создадим декоратор.Это всего лишь ещё один вызов функции
new_decorator = decorator_maker()
Я создаю декораторы! Я буду вызван только раз: когда ты
попросишь меня создать декоратор.
Я возвращаю декоратор.
# Теперь декорируем функцию
def decorated_function():
    print("Я - декорируемая функция.")
decorated_function = new_decorator(decorated_function)
Я - декоратор! Я буду вызван только раз: в момент деко-
рирования функции.
Я возвращаю обёрнутую функцию.
# Теперь вызовем функцию:
decorated_function()
Я - обёртка вокруг декорируемой функции.
Я буду вызвана каждый раз, когда ты вызываешь декориру-
емую функцию.
Я возвращаю результат работы декорируемой функции.
Я - декорируемая функция.
```

Вместо трёх последних строк можно записать:

```
@decorator_maker()
    def decorated_function():
        print("Я - декорируемая функция.")
```

Декоратору можно передавать любые аргументы, как и обычной функции.

Некоторые особенности работы с декораторами:

1) декораторы замедляют вызов функции;

2) если функция декорирована, то декорирование нельзя отменить;

3) декораторы оборачивают функции, что затрудняет отладку.

Последняя проблема частично решена добавлением в модуле `functools` функции `functools.wraps`, копирующей всю информацию об оборачиваемой функции (её имя, из какого она модуля, её документацию и др.) в функцию-обёртку. Декораторы могут быть исполь-

зованы для расширения возможностей функций из сторонних библиотек (код которых мы не можем изменять), или для упрощения отладки (мы не хотим изменять код, который ещё не устоялся). Можно использовать декораторы для расширения различных функций одним и тем же кодом, без повторного его переписывания каждый раз.

2.9 PEP 8 – руководство по написанию кода на Python

В этом разделе кратко описывается соглашение о том, как составлять программы на языке Python, включая стандартную библиотеку, входящую в состав Python. PEP 8 создан на основе рекомендаций Гвидо ван Россума с добавлениями от Барри. Если где-то возникал конфликт, мы выбирали стиль Гвидо. И, конечно, этот PEP может быть неполным (фактически, он, наверное, никогда не будет закончен).

Ключевая идея Гвидо такова: код читается намного больше раз, чем пишется. Собственно, рекомендации о стиле написания кода направлены на то, чтобы улучшить читаемость кода и сделать его согласованным между большим числом проектов. В идеале весь код будет написан в едином стиле, и любой сможет легко его прочесть. Это руководство о согласованности и единстве. Важно писать программы согласно этому руководству. Необходима также согласованность внутри одного проекта, внутри модуля или функции. Но важно помнить, что иногда это руководство неприменимо, и понимать, когда можно отойти от рекомендаций. Две причины для того, чтобы нарушить данные правила:

1. Когда применение правила сделает код менее читаемым даже для того, кто привык читать код, который следует правилам.
2. Чтобы писать в едином стиле с кодом, который уже есть в проекте и который нарушает правила – но это возможность переписать чужой код.

Внешний вид кода. Отступы

Используйте 4 пробела на каждый уровень отступа.

Продолжительные строки должны выравнивать обернутые элементы либо вертикально, используя неявную линию в скобках (круглых, квадратных или фигурных), либо с использованием висячего отступа. При использовании висячего отступа следует применять следующие соображения: на первой линии не должно быть аргументов, а остальные строки должны четко восприниматься как продолжение линии.

Правильно:

```
# Выровнено по открывающему разделителю
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
# Больше отступов включено для отличения его от остальных
def long_function_name(
            var_one, var_two, var_three,
            var_four):
    print(var_one)
```

Неправильно: аргументы на первой линии запрещены, если не используется вертикальное выравнивание.

```
foo = long_function_name(var_one, var_two,
   var_three, var_four)
#Больше отступов требуется,для отличения его от остальных
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Опционально:

```
# Нет необходимости в большем количестве отступов.
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

Закрывающие круглые/квадратные/фигурные скобки в многострочных конструкциях могут находиться под первым непробельным символом последней строки списка, например:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

либо быть под первым символом строки, начинающей многострочную конструкцию:

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

```
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Табуляция или пробелы

Пробелы – самый предпочтительный метод отступов.

Табуляция должна использоваться только для поддержки кода, написанного с отступами с помощью табуляции. Python 3 запрещает смешивание табуляции и пробелов в отступах. Python 2 пытается преобразовать табуляцию в пробелы. Когда Вы вызываете интерпретатор Python 2 в командной строке с параметром -t, он выдает предупреждения (warnings) при использовании смешанного стиля в отступах, а запустив интерпретатор с параметром -tt, Вы получите в этих местах ошибки (errors).

Максимальная длина строки

Ограничьте длину строки максимум 79 символами. Для более длинных блоков текста с меньшими структурными ограничениями (строки документации или комментарии) длину строки следует ограничить 72 символами. Ограничение необходимой ширины окна редактора позволяет иметь несколько открытых файлов бок о бок, и хорошо работает при использовании инструментов анализа кода, которые предоставляют две версии в соседних столбцах. Некоторые команды предпочитают большую длину строки. Для кода, поддерживающегося исключительно или преимущественно этой командой, нормально увеличение длины строки с 80 до 100 символов (фактически увеличивая максимальную длину до 99 символов) при условии, что комментарии и строки документации все еще будут 72 символа.

Стандартная библиотека Python консервативна и требует ограничения длины строки в 79 символов (а строк документации/комментариев в 72). Предпочтительный способ переноса длинных строк является использование подразумеваемых продолжений строк Python внутри круглых, квадратных и фигурных скобок. Длинные строки могут быть разбиты на несколько строк, обернутые в скобки. Это предпочтительнее использования обратной косой черты для продолжения строки.

Пустые строки

Отделяйте функции верхнего уровня и определения классов двумя пустыми строками. Определения методов внутри класса разделяются одной пустой строкой. Дополнительные пустые строки можно использовать для разделения различных групп похожих функций. Пустые строки могут быть опущены между несколькими связанными одно-

строчниками (например, набор фиктивных реализаций). Используйте пустые строки в функциях, чтобы указать логические разделы. Python расценивает символ Ctrl+L как незначащий (whitespace), и Вы можете использовать его, потому что многие редакторы обрабатывают его как разрыв страницы. Таким образом, логические части в файле будут на разных страницах. Но не все редакторы распознают Ctrl+L и на его месте может быть другой символ.

Кодировка исходного файла

Кодировка Python должна быть UTF-8 (ASCII в Python 2).

Файлы в ASCII (Python 2) или UTF-8 (Python 3) не должны иметь объявления кодировки. В стандартной библиотеке нестандартные кодировки должны использоваться только для целей тестирования, либо когда комментарий или строка документации требует упомянуть имя автора, содержащего не ASCII символы. В остальных случаях использование \x, \u, \U или \N – наиболее предпочтительный способ включить символы, не входящие в кодировку ASCII, в строковых литералах. Начиная с версии Python 3.0, в стандартной библиотеке действует следующее соглашение: все идентификаторы обязаны содержать только ASCII символы и означать английские слова везде, где это возможно (во многих случаях используются сокращения или неанглийские технические термины). Кроме того, строки и комментарии тоже должны содержать лишь ASCII символы. Исключения составляют:

а) test case, тестирующий не-ASCII особенности программы;

б) имена авторов. Авторы, чьи имена основаны не на латинском алфавите, должны транслитерировать свои имена в латиницу.

Практические задания

1. На языке Python 3 разработать 2 программы (модули) для обработки одномерных массивов (векторов) согласно варианту, ***используя списки***. Одна программа должна работать с целочисленным вектором, а вторая с вещественным вектором.

Вариант 1.

1. Дан вещественный вектор A(n). Найти число ненулевых элементов в векторе.

2. Если у вещественного вектора A(n) хотя бы один элемент меньше, чем –2, то все отрицательные компоненты заменить их квадратами, оставив все остальные без изменения. В противном случае вектор A умножить на 0,1. На печать выдать исходный и полученный вектора.

Вариант 2.

1. Дан вещественный вектор A(n). Найти количество элементов в векторе, абсолютная величина которых больше 7. На печать выдать исходный вектор, искомое количество элементов.

2. Дан вещественный вектор A(n). Получить новый вектор путем умножения элементов, стоящих за максимальным элементом, на минимальный элемент вектора. На печать выдать исходный вектор, минимальный элемент и полученный вектор.

Вариант 3.

1. Дан вещественный вектор A(n). Найти количество элементов этого вектора, больше среднего арифметического всех его элементов.

2. Дан вещественный вектор A(n). Все элементы вектора, предшествующие первому минимальному элементу, умножить на 10, если элемент минимальный по величине встречается в векторе более чем один раз. В противном случае вектор оставить без изменения.

Вариант 4.

1. Дан вещественный вектор A(n). Подсчитать количество таких i, что A[i] не меньше всех предыдущих элементов вектора (A[1], A[2],..., A[i–1]).

2. Дан вещественный вектор A(n). Поменять местами минимальный и последний элементы вектора.

Вариант 5.

1. Дан целочисленный вектор A(n). Построить вектор B(n), i-й элемент которого равен среднему арифметическому первых i-элементов вектора A:B[i] = (A[1]+...+A[i])/i.

2. Дан целочисленный вектор A(n). Поменять местами максимальный и минимальный элементы вектора.

Вариант 6.

1. Дан целочисленный вектор A(n). Подсчитать, сколько раз встречается в этом векторе максимальное по величине число.

2. Дан целочисленный вектор A(n). Найти наибольшее из четных и количество нечетных чисел вектора. На печать выдать исходный вектор и полученный результат.

Вариант 7.

1. Дан целочисленный вектор A(n). Проверьте, есть ли в нем элементы, равные 0. Если есть, найдите номер первого из них, то есть наименьшее i, при котором A[i] = 0.

2. Дан целочисленный вектор A(2n). Все четные числа, стоящие за максимальным элементом, домножить на минимальный элемент. На печать выдать исходный, полученный вектора, максимальный элемент и его индекс, минимальный элемент.

Вариант 8.

1. Дан вещественный вектор A(n). Найти количество элементов вектора, меньших среднего арифметического всех его элементов.

2. Дан вещественный вектор A(n). Получить новый вектор путем умножения элементов, стоящих перед минимальным элементом, на максимальный элемент вектора. На печать выдать исходный вектор, максимальный, минимальный элементы и полученный вектор.

Вариант 9.

1. Дан целочисленный вектор A(n). Проверьте, есть ли в нем отрицательные элементы. Если есть, найдите наибольшее i, при котором A[i]<0.

2. Дан целочисленный вектор A(n). Построить вещественный вектор B(n), i-й элемент которого равен среднему арифметическому двух соседних элементов вектора A:B[i]=(A[i]+A[i+1])/2, (и B[10]=A[10]).

Вариант 10.

1. Дан целочисленный вектор A(n). Построить вектор B(n), который содержит те же числа, что и вектор A(n), но в котором все отрицательные элементы предшествуют всем неотрицательным.

2. Дан целочисленный вектор A(n). Подсчитайте, сколько раз встречается в этом векторе минимальное по величине число.

Вариант 11.

1. Дан целочисленный вектор A(n). Поменять местами максимальный и первый элементы вектора.

2. Дан целочисленный вектор A(n). Подсчитать количество таких i, что A [i] не больше всех предыдущих элементов вектора (A[1], A[2], ..., A[i–1]).

Вариант 12.

1. Дан целочисленный вектор A(n). Подсчитать количество положительных элементов вектора, предшествующих максимальному элементу.

2. Дан целочисленный вектор A(2n). Если в векторе сумма $S_1 = a_1 + a_2 + ... + a_n$ равна сумме $S_2 = a_{n+1} + a_{n+2} + ... + a_{2n}$, то поменять местами первый и последний элементы вектора. На печать выдать исходный вектор, суммы S_1, S_2, преобразованный вектор.

Вариант 13.

1. Дан целочисленный вектор A(n). Найти сумму отрицательных элементов вектора, следующих за максимальным элементом.

2. Дан целочисленный вектор A(2n). Если в последовательности сумма $S_1 = a_1 + a_2 + ... + a_n$ равна сумме $S_2 = a_{n+1} + a_{n+2} + ... + a_{2n}$, то поменять местами максимальный и минимальный элементы вектора. На печать выдать исходный вектор, суммы S_1, S_2, преобразованный вектор.

Вариант 14.

1. Дан целочисленный вектор A(n). Поменять местами первый отрицательный элемент вектора с последним положительным элементом вектора.

2. Дан целочисленный вектор A(n). Найти наименьшее из четных чисел, входящих в вектор. Определить его индекс и поменять местами с первым элементом. На печать выдать исходный вектор, полученный результат и преобразованный вектор.

Вариант 15.

1. Дан целочисленный вектор A(n). Найти количество положительных элементов, стоящих между минимальным и максимальным элементами вектора.

2. Если у вещественного вектора A(n) хотя бы один элемент меньше, чем 2, то все отрицательные компоненты заменить их квадратами, оставив все остальные без изменения. В противном случае вектор A умножить на 0,5. На печать выдать исходный и полученный вектора.

Вариант 16.

1. Дан вещественный вектор A(n). Найти количество элементов в векторе, абсолютная величина которых меньше 5. На печать выдать исходный вектор, искомое количество элементов.

2. Дан вещественный вектор A(n). Получить новый вектор путем умножения элементов, стоящих за минимальным элементом, на максимальный элемент вектора. На печать выдать исходный вектор, максимальный элемент и полученный вектор.

Вариант 17.

1. Дан вещественный вектор A(n). Найти количество элементов этого вектора, больше среднего арифметического всех его элементов.

2. Дан вещественный вектор A(n). Все элементы вектора, предшествующие первому максимальному элементу, разделить на 2, если элемент максимальный по величине встречается в векторе более чем один раз. В противном случае вектор оставить без изменения. На печать выдать исходный и полученный вектор.

Вариант 18.

1. Дан вещественный вектор A(n). Поменять местами минимальный и первый элементы вектора.

2. Дан вещественный вектор A(n). Подсчитать количество таких i, что A[i] не меньше всех предыдущих элементов вектора (A[1], A[2],..., A[i–1]).

Вариант 19.

1. Дан целочисленный вектор A(n). Поменять местами первый и максимальный элементы вектора.

2. Дан целочисленный вектор A(n). Построить вектор B(n), i-й элемент которого равен среднему арифметическому первых i-элементов вектора A:B[i] = (A[1]+...+A[i])/i.

Вариант 20.

1. Дан целочисленный вектор A(n). Подсчитать, сколько раз встречается в этом векторе минимальное по величине число.

2. Дан целочисленный вектор A(n). Найти наибольшее из нечетных и количество четных чисел вектора.

2. Написать программу на языке Python, создающую из русско-английского словаря англо-русский словарь (***обязательно использовать словари (dict)***). Входные данные берутся из файла input.txt, выходные данные записываются в файл output.txt. Входные данные лексикографически отсортированы. Выходные данные тоже должны быть отсортированы. В выходной файл первым записать полученное количество ***английских слов***. Необходимо, чтобы во входном файле находилось как минимум 5 русских слов, которые имеют несколько английских значений. Слова должны быть подобраны так, чтобы в результате одно английское слово имело несколько русских значений.

3. Написать программу на языке Python, реализующую решение задачи с использованием структурированного типа данных «множество» согласно варианту. Программу просчитать для различных исходных данных.

Вариант 1.

Разработать программу, которая во введенном тексте заменяет все строчные гласные буквы «i», «j», «o» на прописные и подсчитывает количество латинских букв. На печать выдать исходный текст, количество латинских букв и преобразованный текст.

Вариант 2.

Разработать программу, которая присваивает некоторой переменной значение «истина», если во введенном тексте содержатся символы, отличные от букв латинского алфавита и цифр, и «ложь» в обратном случае. Во введенном тексте заменить все гласные на "#". На печать выдать исходный текст, значение логической переменной и преобразованный текст.

Вариант 3.

Разработать программу, которая во введенном тексте заменяет все латинские буквы на символ «*», все гласные буквы на знак «+», и подсчитать количество согласных. На печать выдать исходный текст, количество согласных и преобразованный текст.

Вариант 4.

Разработать программу, которая во введенном тексте удаляет все буквы русского алфавита и подсчитывает количество строчных букв латинского алфавита. На печать выдать исходный текст, количество строчных букв латинского алфавита и преобразованный текст.

Вариант 5.

Разработать программу, присваивающую некоторой переменной значение «истина», если согласных букв во введенном тексте больше, чем гласных букв, и значение «ложь» в обратном случае. На печать выдать исходный текст, промежуточные результаты и значение логической переменной.

Вариант 6.

Разработать программу, которая удаляет все цифры из исходного текста, подсчитывает количество гласных букв и удваивает каждую букву латинского алфавита. На печать выдать исходный текст, количество гласных букв и преобразованный текст.

Вариант 7.

Разработать программу, которая во введенном тексте заменяет все строчные гласные на «*», все прописные согласные на знак «+» и подсчитывает количество цифр. На печать выдать исходный текст, количество цифр и преобразованный текст.

Вариант 8.

Разработать программу, которая во введенном тексте все цифры заменяет на знак пробела, подсчитывает количество прописных гласных и все строчные буквы латинского алфавита заменяет на знак «*». На печать выдать исходный текст, количество прописных гласных и преобразованный текст.

Вариант 9.

Разработать программу, которая во введенном тексте удваивает все прописные гласные буквы, удваивает все цифры и подсчитывает количество пробелов. На печать выдать исходный текст, количество пробелов и преобразованный текст.

Вариант 10.

Разработать программу, которая присваивает некоторой переменной значение «истина», если во введенном тексте содержатся символы, отличные от букв латинского алфавита, цифр и пробела, и значение «ложь» в обратном случае. На печать выдать исходный текст, значение логической переменной.

Вариант 11.

Разработать программу, которая во введенном тексте заменяет все строчные буквы на знак «*», подсчитывает количество прописных со-

гласных и заменяет их на «–». На печать выдать исходный текст, количество прописных согласных и преобразованный текст.

Вариант 12.

Разработать программу, которая во введенном тексте заменяет все знаки «,», «.», «!», «?» на знак «*», все строчные буквы латинского алфавита на «–» и подсчитывает их количество. На печать выдать исходный текст, количество строчных букв латинского алфавита и преобразованный текст.

Вариант 13.

Разработать программу, которая во введенном тексте удаляет все строчные буквы русского алфавита, подсчитывает количество прописных букв латинского алфавита и заменяет все пробелы на знак «#». На печать выдать исходный текст, количество прописных букв латинского алфавита и преобразованный текст.

Вариант 14.

Разработать программу, которая во введенном тексте удаляет все прописные буквы латинского алфавита, подсчитывает количество гласных букв русского алфавита и удваивает каждую цифру. На печать выдать исходный текст, количество гласных букв русского алфавита и преобразованный текст.

Вариант 15.

Разработать программу, присваивающую некоторой переменной значение «истина», если букв латинского алфавита во введенном тексте больше строчных гласных букв русского алфавита, и значение «ложь» в обратном случае. Подсчитать количество цифр. На печать выдать исходный текст, значение логической переменной и количество цифр.

Вопросы для самоконтроля

1. Какими способами можно описать списки в Python?
2. Перечислите основные функции и методы списков.
3. С какого индекса начинается нумерация в списках?
4. У каких специфичных типов данных в Python можно взять элемент по индексу?
5. Что такое кортеж?
6. Зачем применяются кортежи?
7. Какими способами можно описать словари в Python?
8. Перечислите основные методы словарей в Python.
9. Какими способами можно описать множества в Python?
10. Перечислите основные операции над множествами.

11. Перечислите операции, изменяющие множества.

12. Какое основное отличие между set и frosenset?

13. С помощью какой инструкции описывается функция в Python?

14. Как описать функцию с необязательным аргументом?

15. Как описать функцию с переменным количеством позиционных аргументов?

16. С помощью какой инструкции описывается анонимная функция?

17. С помощью какой инструкции можно изменить значение глобальной переменной при изменении значения локальной переменной в функции?

18. Какая функция используется для ввода данных с клавиатуры?

19. Что возвращает функция input()?

20. Что такое декоратор?

21. Перечислите некоторые особенности работы с декораторами в Python.

22. Что такое PEP 8?

ЛИТЕРАТУРА

1. Бизли, Д. М. Python. Подробный справочник / Д. М. Бизли ; пер. с англ. – М. : Символ-Плюс, 2010. – 864 с.

2. Прохоренок, Н. А. PyQt. Создание оконных приложений на Python 3 / Н. А. Прохоренок. – СПб. : БХВ-Петербург, 2011. – 243 с.

3. Лутц, М. Изучаем Python / М. Лутц ; пер. с англ. – 4-е изд. – М. : Символ-Плюс, 2011. – 1271 с.

4. Форсье, Д. Django. Разработка веб-приложений на Python / Д. Форсье, П. Биссекс, У. Чан ; пер. с англ. – М. : Символ-Плюс, 2010. – 456 с.

5. Хахаев, И. А. Практикум по алгоритмизации и программированию на Python / И. А. Хахаев. – М. : ALT Linux, 2010. – 126 с.

6. Чаплыгин, А. Н. Учимся программировать вместе с Питоном / А. Н. Чаплыгин. – М. : Интертраст, 2004. – 110 с.

7. Документация Python [Электронный ресурс]. – Режим доступа: https://pythonworld.ru/. – Дата доступа: 01.06.2022.

Производственно-практическое издание

Кузьменков Дмитрий Сергеевич,
Кузьменкова Екатерина Юрьевна

ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ PYTHON

Практическое пособие

Редактор А. А. Банчук
Корректор В. В. Калугина

Подписано в печать 14.10.2022. Формат 60×84 1/16.
Бумага офсетная. Ризография. Усл. печ. л. 2,8.
Уч.-изд. л. 3,1. Тираж 10 экз. Заказ 485.

Издатель и полиграфическое исполнение:
учреждение образования
«Гомельский государственный университет
имени Франциска Скорины».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий № 3/1452 от 17.04.2017.
Специальное разрешение (лицензия) № 02330 / 450 от 18.12.2013.
Ул. Советская, 104, 246028, Гомель.