Қазақстан Республикасы білім және ғылым министрлігі

Е.А. Бөкетов атындағы Қарағанды университеті

Математика және ақпараттық технологиялар факультеті

Қолданбалы математика және информатика кафедрасы

**Спирина Елена Александровна**

**«Python тілінде программалау (ағылшын тілінде)» пәні бойынша**

**Дәрістер курсы**

білім беру бағдарламасы: «7М01503 Информатика»,

«7М06101 Ақпараттық жүйелер және технологиялар»

Қарағанды 2022

**Theme 1 Introduction to Python**

1. Key points

2. Python features

3. Programming in Python

**1 Key points**

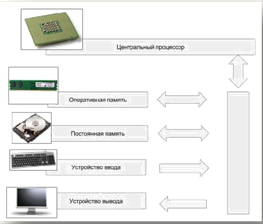
A programmer’s tool is a computer, so let’s briefly look at its device.

All calculations in the computer are performed by the central processor. Files with the program are stored in permanent memory (on the hard disk), and at the time of execution are loaded into temporary (random) memory.

Information is entered into the computer using the keyboard (input device), and output is made using the monitor (output device).

A computer is capable of working with only two types of signals: 1 or 0 (machine code).

Writing programs of the form 1010101010010101010 for a person is difficult, his thinking is different, therefore translator programs have appeared from a programming language that is understandable to humans, to a machine language that is understandable to a computer.



**There are two types of translators:**

1) interpreters;

2) compilers.

The interpreter reads the next program command and immediately executes it without translating the entire program into machine code.

The compiler reads the entire program, makes its translation and creates a finished version of the program in “machine language”, which is then executed.

The Python programming language usually runs in interpretation mode.

As an example of a Python program, we give a program for solving the problem of calculating the area of any rectangle:

*# Entering rectangle sizes*

*a = int (input ('Enter the length of the rectangle in cm'))*

*b = int (input ('Enter the height of the rectangle in cm'))*

*# Calculation of area*

*pl = a \* b*

*# Answer output to the screen*

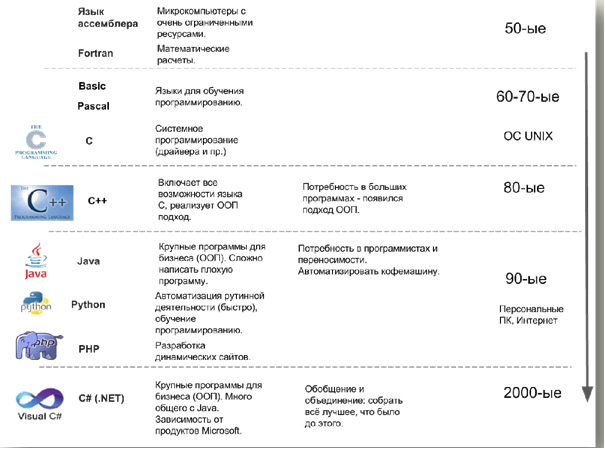
*print ('The area of ​​the rectangle is', pl, 'sq. cm'*

So, Python is a programming language in which programs are developed. To make this process convenient for you and other programmers, programming systems have been developed, including a translator, a text editor (allowing you to copy, delete, and move program fragments), a help system, debugging tools (finding errors in a program), and others. There are several options for the Python language and programming systems for him.

Programming languages that are close to the machine level are called low-level languages (for example, assembly language).

Another type of languages is high-level languages (for example, Python, Java, C #), even more close to human thinking.

Programming languages have an interesting history. They were created not from scratch, but for the specific tasks that their developers had at that time, from which the scope of this or that programming language becomes clear. Today, there are thousands of programming languages, but only a few have played the largest role.



Earlier we said that computer code was the beginning of communication with a computer. Then, in the 50s of the twentieth century, a low-level assembly language appeared, which was closest to the machine level. It is attached to the processor, therefore, its study is equivalent to the study of processor architecture. Programs are written in assembly language today; it is indispensable for small devices (microcontrollers) with very limited memory resources.

The next stage is the appearance of the Fortran language, which was intended for mathematical calculations.

Over time, the need for new staff and the need for training in programming grew. Learning in assembly languages ​​or Fortran required a lot of strength, so in the 60-70s there appeared a galaxy of languages ​​for learning: Basic, Pascal. Pascal is still used in schools as the main language for teaching programming.

At the same time, research is underway in the development of operating systems, which leads to the emergence of a UNIX system. Initially, this operating system was written in assembly language, which complicated its modification and study, then D. Ritchie developed the C language for system programming and, together with B. Kernigan, rewrote the UNIX system in this language. Subsequently, the UNIX operating system became widespread (its GNU / Linux clones are better known today), and with it a lot of programmers appeared for whom the C language became native.

Writing programs in this language requires a good qualification from the programmer, as an undetected error can lead to serious consequences in the program.

Until now, the C language is leading as a language for system programming.

The next stage (the 80s) is characterized by the emergence of object-oriented programming (OOP), which was supposed to simplify the creation of large industrial programs. A scientist appears - B. Straustrup, who lacked the capabilities of the C language, so he expands this language by adding OOP. The new language is called C ++.

In the 90s, personal computers and the Internet appeared; therefore, new technologies and programming languages are required. At this moment, the Java language is gaining popularity, which allows you to start writing large applications in the shortest possible time without any serious damage to the system. The Java language was created with an eye on C ++ and with the promising development of the Internet. This language is characterized by the portability of its programs, i.e. By writing a Java program on a personal computer, you can run it on a coffee machine if there is a Java virtual machine.

Around the same time as Java, Python appears. The language developer, mathematician Guido van Rossum, has been involved in the development of the ABC language for teaching programming for a long time.

With the growth of the Internet, it was necessary to create dynamic sites - the server-side programming language PHP appeared, which today is a leader in the development of websites.

In the 2000s, there has been a tendency for technology to unite around large corporations. At this time, the C # language on the .NET platform was developing.

Python Programming Areas

1. System programming.

2. Development of programs with a graphical interface.

3. Development of dynamic websites.

4. Integration of components.

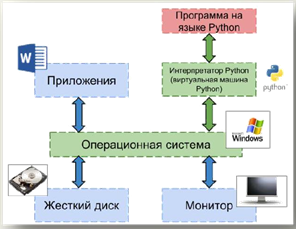
5. Development of programs for working with databases.

6. Rapid prototyping.

7. Development of programs for scientific computing.

8. Game development.

What do we need to run Python programs? Before answering this question, consider how programs run on a computer. The execution of programs is carried out by the operating system (Windows, Linux, etc.). The tasks of the operating system include the allocation of resources (RAM, etc.) for the program, the prohibition or permission to access I / O devices and. etc.



To run Python programs, you need a Python interpreter (virtual machine). This program hides from the Python programmer all the features of the operating system, therefore, writing a program in Python on Windows, you can run it, for example, in GNU / Linux and get the same result.

You can download and install the Python interpreter for free from the official website: http://python.org. To work, we need a Python interpreter version 3 or higher.

Python is one of those rare programming languages that both claim to be simple and powerful.

Officially, Python is represented as follows:

Python is an easy to learn and powerful programming language. It provides efficient, high-level data structures, as well as a simple but effective approach to object-oriented programming. Its elegant syntax and dynamic typing, along with being interpretable, make it an ideal language for scripting and rapid application development in various fields and on most platforms.

**2. Python Features**

1 Simple

Python is a simple and minimal language. Reading a good Python program is very similar to reading an English text, albeit quite strict! This pseudocode nature of Python is one of its greatest strengths. It allows you to focus on solving a problem, not on the language itself.

2 Easy to learn

As you will see, Python is extremely easy to start programming. Python has extremely simple syntax, as noted above.

3 Free and open

Python is an example of free and open source software - FLOSS

(Free / Libré and Open Source Software). Simply put, you have the right to freely distribute copies of this software, read its source, make changes, and use parts of it in your programs. At the heart of free software is the idea of a community that shares its knowledge. This is one of the reasons why Python is so good: it was created and is constantly being improved by a community that just wants to make it better.

4 High level language

When writing a program in Python, you will never have to be distracted by such low-level details as managing the memory used by your program, etc.

5 Portable

Due to its open nature, Python has been ported to many platforms (i.e., modified to work on them). All your programs will be able to run on any of these platforms without any changes, unless you have avoided using system-dependent features.

Python can be used on GNU / Linux, Windows, FreeBSD, Macintosh, Solaris, OS / 2, Amiga, AROS, AS / 400, BeOS, OS / 390, z / OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS , VxWorks, PlayStation, Sharp Zaurus, Windows CE and even PocketPC!

You can even use a platform like Kivy to create games for iOS (iPhone, iPad) and Android.

6 Interpreted

This requires some clarification.

A program written in a compiled programming language, such as C or C ++, is converted from the source language (i.e. C or C ++) to a computer-friendly language (binary code, i.e. zeros and ones) using the compiler using various flags and options. When you run such a program, the linker / loader copies the program from disk to RAM and launches it. Python, by contrast, does not require compilation into binary code. The program is simply executed from source. Python itself converts this source code into some kind of intermediate form called bytecode, and then translates it into machine language and launches it. All this greatly facilitates the use of Python, since there is no need to take care of compiling the program, connecting and loading the necessary libraries, etc. However, this makes Python programs much more portable, as simply copying them to another computer is enough, and they work!

7 Object Oriented

Python supports both procedural and object oriented programming. In procedurally-oriented languages, programs are built on the basis of procedures or functions, which are simply reusable program fragments. In object-oriented programming languages, programs are built on the basis of objects that combine data and functionality. Python provides simple but powerful tools for OOP, especially when compared to large programming languages ​​such as C ++ or Java.

8 Expandable

If you need some critical part of the program to work very fast or you are forced to hide part of the algorithm, you can write this part of the program in C or C ++ and then call it from a program in Python.

9 Built-in

Python can be embedded in C / C ++ programs to provide scripting capabilities to their users.

10 Extensive Libraries

The Python standard library is just huge. It can help in solving a wide variety of tasks related to the use of regular expressions, generating documentation, checking code blocks, parallelizing processes, databases, web browsers, CGI, FTP, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interface) and other system-dependent things. Remember that all this is available absolutely everywhere where Python is installed. This is the Python all-inclusive philosophy.

In addition to the standard library, there are many other high-quality libraries that can be found in the Python Package Catalog

**3. Python programming.**

To familiarize yourself with the development of programs in Python, we divide the process into two stages.

To program in Python, you need the following.

• Create a program by typing and saving it in a file, say, by name myprogram.ru.

• Run (or execute) a program by entering python *myprogram.ру* in the terminal application window.

The Python compiler translates the Python program into a language more suitable for execution on the computer. The Python interpreter then proceeds to control the computer according to these instructions.



**There are two ways to run Python programs:**

1) using the interactive prompt of the interpreter

2) using a file with program text.

***1. Using the interpreter command line***

Open a terminal window (as described in the Installation chapter) and start the Python interpreter by typing python3 and pressing Enter.

Windows users can run the interpreter on the command line if they set the PATH variable appropriately. To open a command prompt in Windows, go to the "Start" menu and click "Run ...". In the dialog box that appears, enter “cmd” and press Enter; you will now have everything you need to get started with python on the DOS command line.

**If you are using IDLE, click Start! "Programs" ! Python 3.0! "IDLE (Python GUI)."**

Once you have started python3, you should see >>> at the beginning of the line where you can type something. This is called the Python interpreter command line. Now type print ('Hello World') and press Enter. As a result, the words “Hello World.

Note that Python returns the result of the string immediately!

You have just entered a single Python “statement”. print is used to print any value passed to it. In this case, we passed the text “Hello World” into it, which was printed on the screen.

***Tip*: How to exit the interpreter command line**

If you are using IDLE or the GNU / Linux or BSD shell, press Ctrl-D or enter exit () and then press Enter.

If you are using the Windows command line, press Ctrl-Z, and then press Enter.

***The Basic literature***

1. Мэтиз Эрик.Изучаем Python. Программирование игр, визуализация данных, веб-приложения. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Библиотека программиста»).

2. [Mark Lutz](http://www.allitebooks.org/author/mark-lutz/). Learning Python. – O’Reilly, 2013.

3. Allen B. Downey. Think Python. - O’Reilly, 2015.

4. Brett Slatkin.  Effective Python: 59 Specific Ways to Write Better Python. - 2015. – 256 p.

5. [Mark Lutz. Programming Python: Powerful Object-Oriented Programming](https://www.amazon.com/Programming-Python-Powerful-Object-Oriented/dp/0596158106/ref=dp_ob_title_bk) O'Reilly Media, 2016. – 1552 p.

6. МакГрат Майк. Программирование на Python для начинающих. – М.: Эксмо, 2015. -  192 с.

***Additional***

7. Саммерфилд М. Программирование на Python 3. Подробное руководство. – Пер. с англ. – СПб.: Символ\_Плюс, 2009. – 608 с., ил.

8. Златопольский Д. М. Основы программирования на языке Python. – М.: ДМК Пресс, 2017. – 284 с.: ил.

***Internet resources***

http://www.python.org/

http://pythontutor.ru/

http://www.enthought.com/products/epdlibraries.php

http://www.edu.ru/modules.php?op=modload&name=Web\_Links&file=index&l\_op=viewlink

https://docs.python.org/3/tutorial/index.html

http://www.python.org/

http://pythontutor.ru/

**Theme 2. Variable and simple data types / Переменные и простые типы данных**

**1. Basic Python Concepts**

**2. Features of working with variables in Python**

**3. Types of data:** Типы данных

**3.1 Numbers** Числа

**3.2 Логический (boolean)** Логические значения

**3.3 String** Строки

**3.4 Lists** Списки

**3.5 Data Type Conversion** Преобразование типов данных

**1. Basic Python Concepts**

Data is stored in computer memory cells. A variable is a name associated with a data type value. Variables are used to track changes in values during the calculation.

**Rules for choosing variable names**

**** Variable names can only consist of letters, numbers, and underscores. They cannot begin with a number. For example, you can name the variable message\_1, but not 1\_message.

 Spaces in variable names are not allowed (underscores can be used). For example, the name greeting\_message is valid, and the name greeting message will throw an error.

 Do not use function names and Python keywords as variable names, such as print

 Variable names should be short but meaningful. For example, name is better than n, student\_name is better than s\_n, and name\_length is better than length\_of\_persons\_name.

It is necessary to introduce some terminology.

**View (consider) a piece of code:**

*a = 1234*

*b = 99*

*c = a + b*

This code creates three objects, each of type int, using the literals 1234 and 99, as well as the expression a + b and binds (bind) (the technical term for creating an association) the variables a, b and c to these objects using operators assignment statement.

As a result, it turns out that the variable c is associated with an object of type int that stores the value 1333.

Python **literal** is a representation of a data type value in code.

A Python **operator** is a representation of a data type operation in code.

A Python **identifier** is a representation of a name in code. Each identifier is a sequence of characters, numbers, and underscores, the first of which cannot be a digit.

**Variable.** A variable is a name associated with a value of a specific data type. Variables are used to track changes in values along the way. Each variable is characterized by a name and type.

An **expression** is a combination of literals, variables, and operators that Python evaluates to get a value.

**2. Features of working with variables in Python**

In a Python program, variables need not be explicitly declared, i.e. the type is identified with the data, and not with the variable.

In Python, variables have no type. But the type is in the data that is "remembered" using variables.

In Python, variables refer to data rather than contain it, as in some other programming languages.

**The object**

All data values ​​in a Python program are represented by objects and relationships between objects.

An object is a representation in machine memory of the value of a particular data type.

Each object is characterized by its identifier, type and value.

**3. Data Types**

The type of an object completely determines its behavior - it is a set of values that it can represent, and a set of operations that can be performed with it.

A data type is a set of values and a set of operations defined for these values.

**Python has five standard types:**

* числа (numbers);
* строки (string);
* списки (list);
* кортежи (tuple);
* словари (dictionary).

**Python has several types of data built in.**

We will consider Python's built-in data types such as **int** (for integers), **long** (for long integers), **float** (for floating-point numbers), **str** (for sequences of characters), and **bool** (for true and false).

Таблица 1. Основные встроенные типы данных

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Набор значений** | **Общие операторы** | **Пример литералов** |
| **int** | Целые числа | + - \* // % \*\* | 99 12 2147483647 |
| **float** | Числа с плавающей точкой | + - \* / \*\* | 3.14 2.5 6.022е23 |
| **bool** | Значения **true** и **false** | **and** или **not** | True False |
| **str** | Последовательности символов | + | 'АВ' 'Hello' '2.5' |

**3.1 Numbers**

To convert numbers from real to integer and vice versa, the functions int () and ﬂoat () are defined in Python.

For example, int (12.6) gives 12 as a result, and ﬂ oat (12) gives 12.0 as a result (the decimal separator is a period**).**

**Type conversion functions (функции преобразования)**

Int ([object], [base of the number system -основание системы счисления]) - conversion to integer.

int("11111111", 2)

255

int(12.6)

12

float([X]) - conversion to a floating point number. If no argument is specified, it returns 0.0.

ﬂoat(12)

12.0

str([object]) - converts the argument to a string

str(10)

'10'

bin() - convert decimal to binary

bin(10)

'0b1010‘

list()  converts the argument to the list:

list("string")

['s', 't', 'r', 'i', 'n', 'g']

list({1,2,3})

[1, 2, 3]

list((1,2,3,4))

[1, 2, 3, 4]

Python supports four different numeric data types:

**int** - integers;

**long** - large integers (can also be used for octal (octal) or decimal (hexadecimal) numbers);

**float** - floating point numbers;

**complex** - complex numbers.

Basic operations with numbers are given in table 2.

Table 2. Operations with numbers

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| x+y | Сложение (сумма x и y) |
| x−y | Вычитание (разность x и y) |
| x∗y | Умножение (произведение x и y) |
| x/y | Деление x на y (частное).  *Внимание!* Если x и y целые, то результат всегда будет целым числом! Для получения вещественного результата хотя бы одно из чисел должно быть вещественным. Пример: 100/8 → 12, а вот 100/8.0 → 12.5 |
| x//y | Целочисленное деление (результат — целое число). Если оба числа в операции вещественные, получается вещественное число с дробной частью, равной нулю. Пример:  100//8 → 12  101.8//12.5 → 8.0 (для сравнения 101.8/12.5 → 8.1440000000000001) |
| x%y | Остаток от целочисленного деления x на y. Пример: 10%4 → 2 |
| x∗∗y | Возведение в степень (x в степени y). Работает и для вещественных чисел. Примеры:  2∗∗3 → 8  2.3∗∗(−3.5) → 0.05419417057580235 |
| −x | Смена знака числа |

**In addition, Python uses built-in functions for number operations:**

**abs ()** (calculation of the absolute value - modulus, abs (−3) → 3),

**pow ()** (exponentiation, pow (2,3) → 8),

**divmod ()** (calculation of the result of integer division and remainder,

**divmod (17,5) → (3,2))**

**round () (rounding, round (100.0 / 6) → 17.0).**

All other functions for working with numbers (mathematical), such as calculating the square root, sine, etc., require the connection of the **math module.**

**3.2 Boolean values.**

**Boolean (**Logical) values in Python are represented by two quantities - the logical constants True (True) and False (False).

Logical values are obtained as a result of logical operations and the calculation of logical expressions.

Таблица 3. Основные логические операции и выражения

|  |  |
| --- | --- |
| **Операция или выражение** | **Описание** |
| > | Условие «больше» (например, проверяем, что a > b) |
| < | Условие «меньше» (например,проверяем, что a < b) |
| == | Условие равенства (проверяем, что a равно b) |
| != | Условие неравенства (проверяем, что a не равно b) |
| not x | Отрицание (условие x не выполняется) |
| x and y | Логическое «И» (умножение). Чтобы выполнилось условие x and y, необходимо, чтобы одновременно выполнялись условия x и y. |
| x or y | Логическое «ИЛИ» (сложение). Чтобы выполнилось условие x or y, необходимо, чтобы выполнилось одно из условий. |
| x in A | Проверка принадлежности элемента x множеству (структуре) A (см. «Структуры данных»). |
| a < x < b | Эквивалентно (x > a) and (x < b) |

**3.3 Strings**

Strings (sequences of characters - letters and other icons) can consist of English characters and any other alphabet.

In Python, strings and characters must be enclosed in quotation marks (single or double).

Elements (characters) in a line are numbered starting from zero.

A single character - a letter - is “from the point of view of Python” a string consisting of one element. The maximum number of characters per line (line length) in Python is limited only by the available memory. So text of any reasonable size (for example, several thousand pages) can be written in one line of Python.

Numbers can be converted to strings using the **str ()** function.

Таблица 4. Основные операции со строками

|  |  |
| --- | --- |
| **Функция или операция** | **Описание и результат** |
| len(s) | **Вычисляется длина строки s как число символов** |
| s1+s2 | **Конкатенация**. К концу строки **s1** присоединяется строка **s2**, в результате получается новая строка, например, ’вы’+’года’→’выгода’ |
| s∗n (или n∗s) | **n**-кратное повторение строки **s**, в результате получается новая строка, например ’кан’∗2→’канкан’ |
| s[i] | **Выбор из s элемента с номером i,** нумерация начинается с 0 (первый элемент имеет номер 0). Результатом является символ. Если **i<0**, отсчёт идёт с конца (первый символ строки имеет номер 0, последний имеет номер−1).  Пример:  s=’дерево’  s[2]→’р’  s[−2]→’в’ |
| s[i:j:k] | **Срез — подстрока, содержащая символы строки s с номерами от i до j с шагом** **k** (элемент с номером **i** входит в итоговую подстроку, а элемент с номером **j** уже не входит). Если **k** не указан (использован вариант **s[i:j]**), то символы идут подряд (равносильно **s[i:j:1]**).  Примеры:  s=’derevo’  s[3:5]→’ev’  s[1:5:2]→’ee’ |
| min(s) | **Определяет и выводит (возвращает) символ с наименьшим значением** (кодом – номером в кодовой таблице)  Пример:  s=’derevo’  min(s)→’d’ |
| max(s) | **Возвращает символ с наибольшим значением** (кодом)  Пример:  s=’derevo’  max(s)→’v’ |

Strings, like Python objects, have methods (i.e. functions that the objects themselves execute). The main methods are listed in table 5.

Let the string to which these methods apply is called s1.

Table 5. String Methods

|  |  |
| --- | --- |
| **Метод** | **Описание и результат** |
| s1.center(n) | Возвращается строка **s1**, **дополненная пробелами справа и слева до ширины** в **n** символов. Исходная строка не изменяется. Если **n≤len(s1)**, пробелы не добавляются.  Пример:  s1=’Zoom−Zoom’  s1.center(15) → ’␣␣␣Zoom−Zoom␣␣␣’ |
| s1.ljust(n) | Строка **s1** **выравнивается по левому краю** (дополняется пробелами справа) в пространстве шириной **n** символов. Если **n< len(s1)**, пробелы не добавляются. Пример:  s1=’Zoom−Zoom’  s1.ljust(15)→’Zoom−Zoom␣␣␣␣␣␣’ |
| s1.rjust(n) | Строка **s1** **выравнивается по правому краю** (дополняется пробелами слева) в пространстве шириной **n** символов. Если **n< len(s1)**, пробелы не добавляются. Пример:  s1=’Zoom−Zoom’  s1.rjust(15)→’␣␣␣␣␣␣Zoom−Zoom’ |
| s1.count(s[, i,j]) | **Определяется количество вхождений подстроки s в строку s1.** Результатом является число. Можно указать позицию начала поиска **i** и окончания поиска **j** (по тем же правилам, что и начало и конец среза).  Примеры:  s1=’abrakadabra’  s1.count(’ab’)→2  s1.count(’ab’,1)→1  s1.count(’ab’,1,−3)→0, потому что s1[1:−3]→’brakada’ |
| s1.ﬁnd(s[, i,j]) | **Определяется позиция первого (считая слева) вхождения подстроки s в строку s1.** Результатом является число. Необязательные аргументы **i** и **j** определяют начало и конец области поиска (как в предыдущем случае).  Пример:  s1=’abrakadabra’  s1.ﬁnd(’br’)→1 |
| s1.rﬁnd(s[, i,j]) | **Определяется позиция последнего (считая слева) вхождения подстроки s в строку s1.** Результатом является число. Необязательные аргументы **i** и **j** определяют начало и конец области поиска (как в предыдущем случае). Пример:  s1=’abrakadabra’  s1.rﬁnd(’br’)→8 |
| s1.strip() | **Создаётся копия строки, в которой удалены пробелы в начале и в конце** (если они есть или образовались в результате каких-то операций).  Пример:  s1=’␣breKeKeKeKs␣’  s2=s1.strip()  s2→’breKeKeKeKs’ |
| s1.lstrip() | **Создаётся копия строки, в которой удалены пробелы в начале** (если они есть или образовались в результате каких-то операций).  Пример:  s1=’␣breKeKeKeKs␣’  s2=s1.lstrip()  s2→’breKeKeKeKs’ |
| s1.rstrip() | **Создаётся копия строки, в которой удалены пробелы в конце** (если они есть или образовались в результате каких-то операций).  Пример:  s1=’␣breKeKeKeKs␣’  s2=s1.rstrip()  s2 → ’breKeKeKeKs’ |
| s1.replace(s2,s3[,n]) | **Создаётся новая строка, в которой фрагмент (подстрока) s2 исходной строки заменяется на фрагмент s3.** Необязательный аргумент **n** указывает количество замен (если требуется заменить не все фрагменты).  Пример:  s1=’breKeKeKeKs’  ss=s1.replace(’Ke’,’XoXo’,2)  ss→’breXoXoXoXoKeKs’ |
| s1.capitalize()  или  s1.title() | **Создаётся новая строка, в которой первая буква исходной строки становится заглавной** (прописной), а все остальные становятся маленькими (строчными). Пример:  s1=’breKeKeKeKs’  s2=s1.capitalize()  s2→’Brekekekeks’ |
| s1.swapcase() | **Создаётся новая строка, в которой прописные буквы исходной строки заменяются на строчные и наоборот**.  Пример:  s1=’breKeKeKeKs’  s2=s1.swapcase()  s2→’BREkEkEkEkS’ |
| s1.upper() | **Создаётся новая строка, в которой все буквы исходной строки становятся заглавными** (прописными).  Пример:  s1=’breKeKeKeKs’  s2=s1.upper()  s2 → ’BREKEKEKEKS’ |
| s1.lower() | **Создаётся новая строка, в которой все буквы исходной строки становятся маленькими** (строчными).  Пример:  s1=’breKeKeKeKs’  s2=s1.lower()  s2→’brekekekeks’ |

**3.4 Lists**

A list is a collection of items that follow in a specific order.

In Python, a list is indicated by square brackets ([]), and individual list items are separated by commas. A simple example of a list with bike model names:

bicycles=['trek', 'cannondale', 'redline', 'specialized']

print(bicycles)

результат

['trek', 'cannondale', 'redline', 'specialized']

***Tuples (кортеж)* -** data type in Python, similar to lists and contains elements separated by commas.

  Unlike lists, elements in a tuple are limited to parentheses ().

***The main difference between lists (lists) and tuples (tuples):*** *lists are limited to square brackets ([ ]) and their elements and size can be changed, while tuples are limited to parentheses ( ) and cannot be changed.*

**Dictionaries (словари)** in Python are a kind of hash tables.

They are also similar to associative arrays and hashes in Perl, and contain *key:value* pairs.

The key in the dictionary can contain any data type used in Python, but, as a rule, these are numbers or strings.

Values can be any random elements.

Dictionaries are enclosed in curly brackets ({}), and access to the values of elements is done using square brackets ([]).

**3.5 Data Type Conversion**

Python's built-in data type conversion functions return a new object representing the converted value.

The most commonly used of these functions are presented in the table below.



***The Basic literature***

1. Мэтиз Эрик.Изучаем Python. Программирование игр, визуализация данных, веб-приложения. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Библиотека программиста»).

2. [Mark Lutz](http://www.allitebooks.org/author/mark-lutz/). Learning Python. – O’Reilly, 2013.

3. Allen B. Downey. Think Python. - O’Reilly, 2015.

4. Brett Slatkin.  Effective Python: 59 Specific Ways to Write Better Python. - 2015. – 256 p.

5. [Mark Lutz. Programming Python: Powerful Object-Oriented Programming](https://www.amazon.com/Programming-Python-Powerful-Object-Oriented/dp/0596158106/ref=dp_ob_title_bk) O'Reilly Media, 2016. – 1552 p.

6. МакГрат Майк. Программирование на Python для начинающих. – М.: Эксмо, 2015. -  192 с.

***Additional***

7. Саммерфилд М. Программирование на Python 3. Подробное руководство. – Пер. с англ. – СПб.: Символ\_Плюс, 2009. – 608 с., ил.

8. Златопольский Д. М. Основы программирования на языке Python. – М.: ДМК Пресс, 2017. – 284 с.: ил.

***Internet resources***

http://www.python.org/

http://pythontutor.ru/

http://www.enthought.com/products/epdlibraries.php

http://www.edu.ru/modules.php?op=modload&name=Web\_Links&file=index&l\_op=viewlink

https://docs.python.org/3/tutorial/index.html

http://www.python.org/

http://pythontutor.ru/

**Theme 3 Conditional instructions**

1. Branching with a conditional statement

2. Template of the if statement

3. Pattern of conditional operator if: elif: else

**1 Branching using a conditional statement**

The term control flow is used to describe a change in the order of operators in a program.

Python conditional statements implement conditional expressions, where some other statements may or may not be executed depending on certain conditions.

**2. The template of the if statement.**

In Python, the statement specified by the if keyword performs a condition check that evaluates the given expression for True or False values.

This allows the program to continue actions in various directions, depending on the result of this test.

This procedure is known as conditional branching.

If template



As a second example, consider the following code:

**if x> y:**

**temp = x**

**x = y**

**y = temp**

This code arranges the values of the variables x and y in ascending order, exchanging them with references to objects if necessary.

****

**In Python, the size of the indentation of each line is important.**

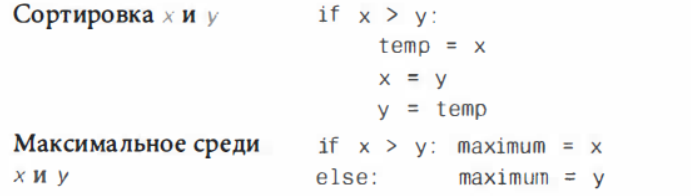
The if statement code on the left has a block with one statement followed by another statement; The if statement code on the right has a block with two statements.

If the value of x is greater than or equal to 0, then both fragments, not and negative, are output.

In contrast, if x is less than 0, the code on the left only outputs negative, and the code on the right does not display anything.

Using the else keyword, you can modify this construct by adding instructions to execute in case the validation was not successful.



****

**3. The conditional statement template if: elif: else**

After the if check block, the elif (else if) keyword can also be used, which involves ruuning an alternative check and following the corresponding instructions.

a = **3**

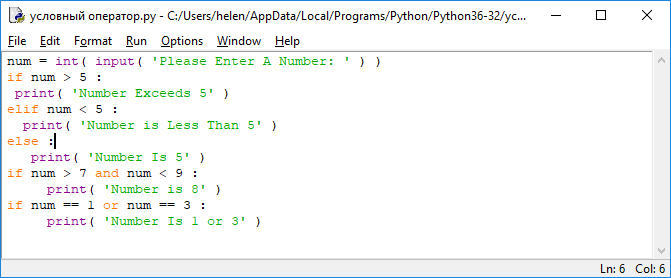
if a > **2**:

print("H")

else:

print("L")

**Program.** Enter an integer. If the number is more than 5 - output “Number is more than 5”, if less, then “Number is less than 5”, if it is equal, then “Number is 5”



***The Basic literature***

1. Мэтиз Эрик.Изучаем Python. Программирование игр, визуализация данных, веб-приложения. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Библиотека программиста»).

2. [Mark Lutz](http://www.allitebooks.org/author/mark-lutz/). Learning Python. – O’Reilly, 2013.

3. Allen B. Downey. Think Python. - O’Reilly, 2015.

4. Brett Slatkin.  Effective Python: 59 Specific Ways to Write Better Python. - 2015. – 256 p.

5. [Mark Lutz. Programming Python: Powerful Object-Oriented Programming](https://www.amazon.com/Programming-Python-Powerful-Object-Oriented/dp/0596158106/ref=dp_ob_title_bk) O'Reilly Media, 2016. – 1552 p.

6. МакГрат Майк. Программирование на Python для начинающих. – М.: Эксмо, 2015. -  192 с.

***Additional***

7. Саммерфилд М. Программирование на Python 3. Подробное руководство. – Пер. с англ. – СПб.: Символ\_Плюс, 2009. – 608 с., ил.

8. Златопольский Д. М. Основы программирования на языке Python. – М.: ДМК Пресс, 2017. – 284 с.: ил.

***Internet resources***

http://www.python.org/

http://pythontutor.ru/

http://www.enthought.com/products/epdlibraries.php

http://www.edu.ru/modules.php?op=modload&name=Web\_Links&file=index&l\_op=viewlink

https://docs.python.org/3/tutorial/index.html

http://www.python.org/

http://pythontutor.ru/

**4 Theme. Cycle instructions**

1. While loop

2. Loop for in.

3. Operators break / continue

**A loop (цикл)** is a piece of code in a program that automatically repeats itself.

One complete execution of instructions within a loop is called iteration (итерация) or passage (проход).

The cycle size is controlled by the test condition created inside the cycle.

*The loop continues until the test expression is True, and ends at the point where it becomes equal to False.*

****

**1. While loop**

**The while statement implements a loop.**

A block of statements indented within the while loop is the body of the loop, and a logical expression is the loop-continuation condition.

Each while loop is usually preceded by an initialization code that sets the initial value (s).



If the result of the boolean expression is False, do nothing; if the result is True, execute the statement block below.

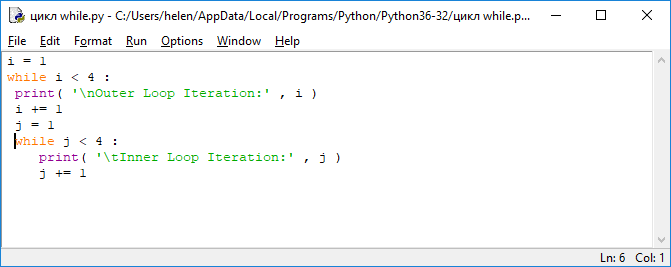
and then check the expression again and execute the sequence of statements if the expression returns True, and continue as long as the expression remains true.

**Nested loops**

Cycles can be nested one into the other, with all iterations of the inner cycle already completed at each iteration of the outer cycle.

For convenience, you can initialize the so-called counter variable before each definition of the cycle, setting its initial value and increasing it at each iteration, as well as including this variable in the test expression.

The loop will end if the expression fails.



**2. The for in loop.**

**For in** loop pattern:

**for element in listname:**

**executable-instructions-at-each-iteration**

**executable-instructions-at-each-iteration**

Using the for in statement, you can loop through list, tuple, set, and dictionary keys.

Since a string is nothing more than a list of characters, each character of a string can be looped around using the for in statement.

Using the for in loop, you can bypass the elements of any list or line characters in the order in which they appear, but you cannot explicitly specify the number of iterations of the loop, the stop condition, or the size of the iteration step.

However, you can use the **range ()** language function to generate a sequence of numbers used for iterations.

**Function range () - generates a sequence starting from zero and ending with a number in brackets, not including it.**

For example, range (5) generates the sequence 0,1,2,3,4.

You can specify two numbers separated by a comma as the parameters of the range () function - the start and end values. For example, range (1,5) generates a sequence of 1,2,3,4. You can also use another version of the function, which takes three parameters, separated by a comma - the initial value, final value and step.

**Enumerate ()** function - displays all indexes and associated values, specifying a list name as a parameter.

**The zip ()** function allows you to traverse several lists at once, specifying list names as parameters separated by commas, and at the output you will receive pairwise values ​​of an element with the same index, separated by commas.

When traversing dictionary elements, you can output key: value pairs using the items () dictionary method and specifying two parameters after the for keyword - one for the key name and one for its value.

**3. Break / continue statements**

In order to force exit the loop when some condition is met, the **break** keyword is used.

It is located inside the loop statement after the expression being checked.

When the check returns **True**, the loop immediately ends and the program transfers control to the next instruction.

For example, if **break** is in a nested inner loop, then control will be passed to the next iteration of the outer loop.

**Task**

Create a loop that runs three times.

Specify a nested *inner loop (internal), внутренний цикл)* that also runs three times.

Organize the output of the counters (both the inner loop and the *outer (external) loop – внешний цикл*) for each pass of the inner loop.

Organize a way out of the inner loop provided

  i == 2 and j == 1

****

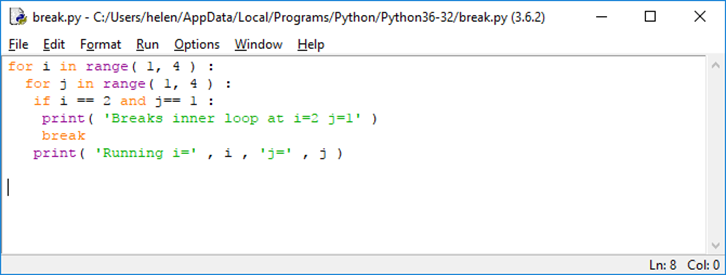
**Task**

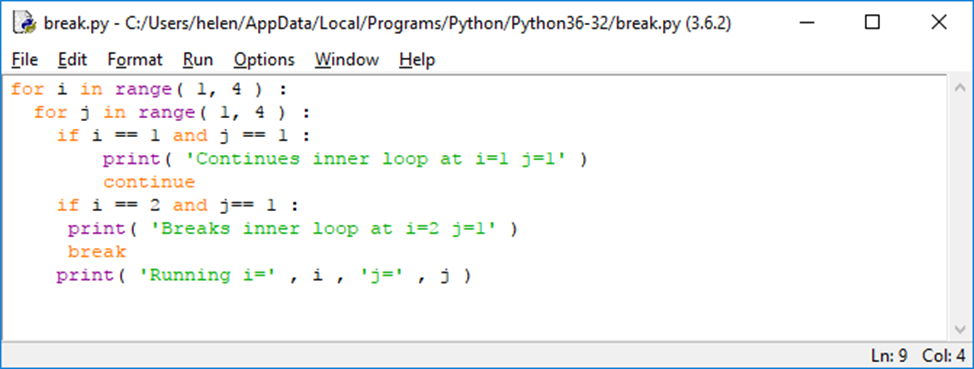
Create a loop that runs three times.

1. Specify a *nested inner loop* that also runs three times.

2. Organize the output of counters (both internal and external) for each pass of the internal cycle.

3. Organize skip the first iteration of the inner loop

****

****

***The Basic literature***

1. Мэтиз Эрик.Изучаем Python. Программирование игр, визуализация данных, веб-приложения. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Библиотека программиста»).

2. [Mark Lutz](http://www.allitebooks.org/author/mark-lutz/). Learning Python. – O’Reilly, 2013.

3. Allen B. Downey. Think Python. - O’Reilly, 2015.

4. Brett Slatkin.  Effective Python: 59 Specific Ways to Write Better Python. - 2015. – 256 p.

5. [Mark Lutz. Programming Python: Powerful Object-Oriented Programming](https://www.amazon.com/Programming-Python-Powerful-Object-Oriented/dp/0596158106/ref=dp_ob_title_bk) O'Reilly Media, 2016. – 1552 p.

6. МакГрат Майк. Программирование на Python для начинающих. – М.: Эксмо, 2015. -  192 с.

***Additional***

7. Саммерфилд М. Программирование на Python 3. Подробное руководство. – Пер. с англ. – СПб.: Символ\_Плюс, 2009. – 608 с., ил.

8. Златопольский Д. М. Основы программирования на языке Python. – М.: ДМК Пресс, 2017. – 284 с.: ил.

***Internet resources***

http://www.python.org/

http://pythontutor.ru/

http://www.enthought.com/products/epdlibraries.php

http://www.edu.ru/modules.php?op=modload&name=Web\_Links&file=index&l\_op=viewlink

https://docs.python.org/3/tutorial/index.html

http://www.python.org/

http://pythontutor.ru/

**Theme 5 Lists**

1. The concept of the list.

2. Work with lists

**1. The concept of a list.**

A list is a collection of items that follow in a specific order.

Any information can be placed in the list, and the data in the list do not even have to be somehow connected with each other. Since a list usually contains more than one element, it is recommended that you assign plural names to letters: letters, digits, names, etc.

In Python, a list is indicated by square brackets ([]), and individual list items are separated by commas.

A simple example of a list with bike model names:

*nums = [0, 1, 2, 3, 4, 5]*

*bicycles = ['trek', 'cannondale', 'redline', 'specialized']*

*print (bicycles)*

*A list of list items in square brackets appears on the screen: ['trek', 'cannondale', 'redline', 'specialized']*

Data is stored sequentially in the "elements" of the list, which are indexed by numerical values, starting from zero. Lists can have more than one index, that is, be multidimensional.

Lists containing three or more indexes are not widespread, but, for example, two-dimensional ones are very often used to store pairs of X, Y coordinates.

The list of string quantities can also be considered multidimensional, since each string is itself a list of characters.

Therefore, each character can be accessed by its numerical index inside a certain string.

****

**Task** 1.

Create a list of three elements containing string values. Организуйте вывод значений, содержащихся в каждом элементе списка

*quarter = [ 'January' , 'February' , 'March' ]*

*print( 'First Month :' , quarter[0] )*

*print( 'Second Month :' , quarter[1] )*

*print( 'Third Month :' , quarter[2] )*

**Task 2.**

**С**оздайте многомерный список из двух элементов, каждый из которых сам по себе является списком из трех элементов, содержащих целые значения.

*coords = [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ]*

Организуйте вывод значений, содержащихся в двух определенных элементах внутреннего списка.

*print( '\nTop Left 0,0 :' , coords[0][0] )*

*print( 'Bottom Right 1,2 :' , coords[1][2] )*

Выведите только ***один символ строковой переменной* quarter**

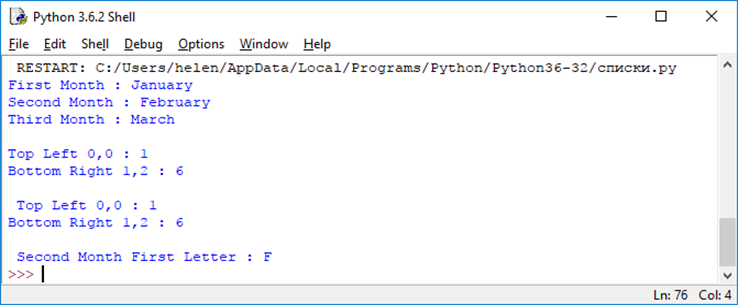
*quarter = [ 'January', 'February', 'March' ]*

*0 [ 'January',*

*1 'February',*

*2 'March' ]*

*print( '\nSecond Month First Letter :' , quarter[1][0] )*

****

**sep =' \* '** – вставляет знак \* между выводимыми значениями

**\n** – пропуск строки

**end='!\n'** добавляет как восклицательный знак, так и символ новой строки

**lang = input( 'Favorite programming language? : ' )**

**print( lang , 'Is' , 'Fun' , sep = ' \* ' , end = '!\n' )**

Favorite programming language? : ENGL

ENGL \* Is \* Fun!

**2. Work with lists**

Lists containing multiple data items are widely used in Python programming.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Table 1. Основные операции со списками   |  |  | | --- | --- | | **Функция или операция** | **Описание и результат** | | len(lst) | Определяется количество элементов списка **lst**. Результат — число. | | lst1+lst2 | Объединение списков. Получается новый список, в котором после элементов списка **lst1** находятся элементы списка **lst2**.  Пример:  lst1=[1,2,3]  lst2=[’raz’,’dva’]  lst3=lst1+lst2  lst3→ [1, 2, 3, ’raz’, ’dva’] | | lst∗n (или n∗lst) | **n**-кратное повторение списка **lst**. Результат — новый список.  Пример:  lst2=[’raz’,’dva’]  lst2∗3→[’raz’,’dva’,’raz’,’dva’,’raz’,’dva’] | | lst[i] | Выбор из **lst** элемента с номером **i**, нумерация начинается с 0 (первый элемент имеет номер 0) Если **i<0**, отсчёт идёт с конца (последний элемент списка имеет номер −1).  Пример:  lst3=[1, 2, 3, ’raz’, ’dva’]  lst3[2]→3  lst3[−2]→’raz’ | | lst[i:j:k] | Срез — список, содержащий элементы списка **lst** с номерами от **i** до **j** с шагом **k** (элемент с номером **i** входит в итоговый список, а элемент с номером **j** уже не входит). Если **k** не указан (использован вариант **lst[i:j]**), то символы идут подряд (равносильно **lst[i:j:1]**).  Пример:  lst3=[1, 2, 3, ’raz’, ’dva’]  lst3[1:4]→[2, 3, ’raz’] | | min(lst) | Определяется элемент с наименьшим значением в соответствии с алфавитным («словарным») порядком.  Пример:  lst3=[1, 2, 3, ’raz’, ’dva’]  min(lst3)→1 | | max(lst) | Определяется элемент с наибольшим значением в соответствии с алфавитным («словарным») порядком.  Пример:  lst3=[1, 2, 3, ’raz’, ’dva’]  max(lst3)→’raz’ | | lst[i]=x | Замена элемента списка с номером i на значение x. Если x является списком, то на место элемента списка будет вставлен список. При этом новый список не создаётся.  Примеры:  lst3=[1, 2, 3, ’raz’,’dva’]  lst3[2]=’ tri ’  lst3→[1, 2, ’tri ’, ’raz’, ’dva’]  lst3[2]=[7,8]  lst3→[1, 2, [7, 8], ’raz’, ’dva’] | | del lst[i] | Удаление из списка элемента с номером **i**. Новый список не создаётся.  Пример:  lst3=[1, 2, [7, 8], ’raz’, ’dva’]  del lst3[2]  lst3→[1, 2, ’raz’, ’dva’] | | lst [i:j]=x | Замена среза списка **lst** на элемент или список **x** (несколько элементов заменяются на **x**).  Примеры:  lst3=[1, 2, 3, ’raz’, ’dva’]  lst3[2:4]=’tri’  lst3→[1, 2, ’t’, ’r’, ’i’, ’dva’]  lst3[2:4]=’a’  lst3→[1, 2, ’a’, ’i’, ’dva’]  *Обратите внимание, что строка интерпретируется как список!* | | del lst[i:j] | Удаление элементов, входящих в указанный срез («вырезание среза»).  Пример:  lst3=[1, 2, ’a’, ’i’, ’dva’]  del lst3[2:4]  lst3→[1, 2, ’dva’] | |

Lists in Python, like strings, are objects, so there are methods for lists that can be accessed through point notation.

Table 2. Основные методы списков

|  |  |
| --- | --- |
| **Метод** | **Описание и результат** |
| lst.append(x) | **Добавление элемента x в конец списка lst**. **x** не может быть списком. Создания нового списка не происходит.  Пример:  lst=[’raz’,’dva’,’tri’,1,2]  lst.append(3)  lst→[’raz’,’dva’,’tri’,1,2,3] |
| lst.extend(t) | **Добавление кортежа или списка t в конец списка lst** (похоже на объединение списков, но создания нового списка не происходит).  Пример:  lst1=[1,2,3]  lst2=[’raz’,’dva’]  lst1.extend(lst2)  lst1→[1,2,3,’raz’,’dva’] |
| lst.count(x) | **Определение количества элементов, равных** **x**, в списке **lst**. Результат является числом.  Пример:  lst=[1,2,3,’raz’,’dva’,’raz’,’dva’]  lst.count(’raz’)→2 |
| lst.index(x) | **Определение первой слева позиции элемента x в списке** **lst**. Если такого элемента нет, возникает сообщение об ошибке.  Пример:  lst=[1,2,3,’raz’,’dva’,’raz’,’dva’]  lst.index(’dva’)→4 |
| lst.remove(x) | **Удаление элемента x в списке lst в первой слева позиции**. Если такого элемента нет, возникает сообщение об ошибке.  Пример:  lst=[1,2,3,’raz’,’dva’,’raz’,’dva’]  lst.remove(’dva’)  lst→[1,2,3,’raz’,’raz’,’dva’] |
| lst.pop(i) | **Удаление элемента с номером i из списка** **lst**. При этом выдаётся значение этого элемента («извлечение» элемента из списка). Если номер не указан, удаляется последний элемент. Новый список не создаётся.  Примеры:  lst=[1,2,3,’raz’,’raz’,’dva’]  \lstinline lst.pop(3)|→’raz’|  lst→[1,2,3,’raz’,’dva’]  \lstinline lst.pop()|→’dva’|  lst→[1,2,3,’raz’] |
| lst.insert (i,x) | **Вставка элемента или списка x в позицию i списка** **lst**. Если **i>=0**, вставка идёт в начало списка. Если **i>len(lst)**, вставка идёт в конец списка. Новый список не создаётся.  Пример:  lst=[1,2,3,’raz’]  lst.insert(3,’tri’)  lst→[1,2,3,’tri’,’raz’] |
| lst.sort() | **Сортировка списка по возрастанию** (в алфавитном порядке). Новый список не создаётся.  Пример:  lst=[1,2,3,’tri’,’raz’]  lst.sort()  lst→[1,2,3,’raz’,’tri’] |
| lst.reverse() | **Замена порядка следования элементов на обратный**. Новый список не создаётся.  Пример:  lst=[1,2,3,’raz’,’tri’]  lst.reverse()  lst→[’tri’,’raz’,3,2,1] |

**Task 3**

1. Create two lists of three elements each containing string values. Print the contents of the first list and its length.

*basket = ['Apple', 'Bun', 'Cola']*

*crate = ['Egg', 'Fig', 'Grape']*

*print ('Basket List:', basket)*

*print ('Basket Elements:', len (basket))*

2. Add an item to the list. Print all the items on this list. Then remove the last item and print all the items in the list.

*basket.append ('Damson')*

*print ('Appended:', basket)*

*print ('Last Item Removed:', basket.pop ())*

*print ('Basket List:', basket)*

3. Add all the elements of the second list to the first list. Print all the elements of the first list, then remove elements 1 through 3. Print the elements of the first list again.

*basket.extend (crate)*

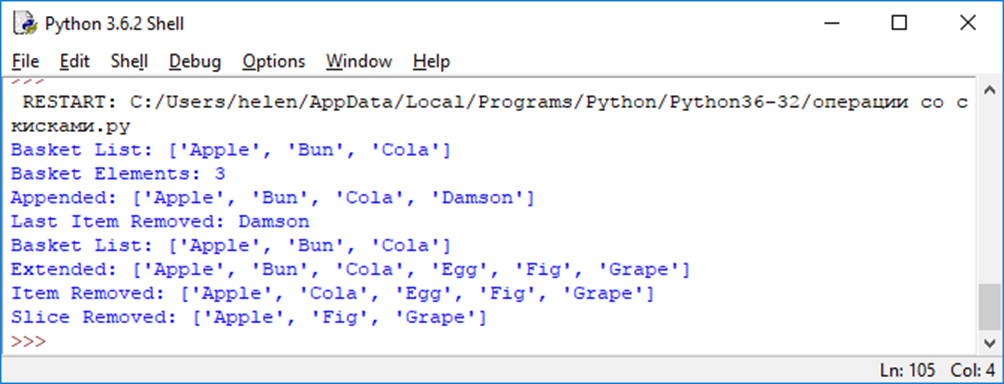
*print ('Extended:', basket)*

*del basket [1]*

*print ('Item Removed:', basket)*

*del basket [1: 3]*

*print ('Slice Removed:', basket)*

****

**Summary**

* In Python, it is possible to initialize several variables at once in a single instruction using multiple assignments.
* Lists are variables that can store multiple data items in numbered items whose index starts from zero.
* The data stored in the list item can be accessed using the list name and the element index in square brackets.

***The Basic literature***

1. Мэтиз Эрик.Изучаем Python. Программирование игр, визуализация данных, веб-приложения. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Библиотека программиста»).

2. [Mark Lutz](http://www.allitebooks.org/author/mark-lutz/). Learning Python. – O’Reilly, 2013.

3. Allen B. Downey. Think Python. - O’Reilly, 2015.

4. Brett Slatkin.  Effective Python: 59 Specific Ways to Write Better Python. - 2015. – 256 p.

5. [Mark Lutz. Programming Python: Powerful Object-Oriented Programming](https://www.amazon.com/Programming-Python-Powerful-Object-Oriented/dp/0596158106/ref=dp_ob_title_bk) O'Reilly Media, 2016. – 1552 p.

6. МакГрат Майк. Программирование на Python для начинающих. – М.: Эксмо, 2015. -  192 с.

***Additional***

7. Саммерфилд М. Программирование на Python 3. Подробное руководство. – Пер. с англ. – СПб.: Символ\_Плюс, 2009. – 608 с., ил.

8. Златопольский Д. М. Основы программирования на языке Python. – М.: ДМК Пресс, 2017. – 284 с.: ил.

***Internet resources***

http://www.python.org/

http://pythontutor.ru/

http://www.enthought.com/products/epdlibraries.php

http://www.edu.ru/modules.php?op=modload&name=Web\_Links&file=index&l\_op=viewlink

https://docs.python.org/3/tutorial/index.html

http://www.python.org/

http://pythontutor.ru/

**Theme 6 Functions**

1. Creating a ***custom /*** **user/ *user-defined function*** (пользовательской функции)

2. Variable scope: Global and local variables

3 Substitution and transfer of arguments - Подстановка и передача аргументов

3.1 Positional arguments - Позиционные аргументы

3.2. Multiple function calls Многократные вызовы функций

3.3. Named Arguments Именованные аргументы

4. Return values - return

1**. Creating a custom function**

Most Python programs can contain a significant number of user-defined functions that are called as needed.

**Function** - a named block of code designed to solve one specific problem.

You can create a custom function using the def (definition) keyword, followed by the function name and brackets of your choice.

As a name for his function, the programmer can choose any identifier with the exception of Python keywords, as well as existing names of built-in functions.

The line with the function definition must end with a colon.

The instructions that must be executed when the function is called (function body) are located on the lines below using indentation.

The syntax for defining a function is as follows:

**def function-name ():**

**executable expression**

**executable expression**

***A function may or may not return a result*** (in the latter case, it would make sense to talk about a procedure - but in Python everything is called a function in one word).

*If a function does not return a result*, then it can be viewed as a sequence of commands that are executed at the same place and time as the function call.

In the body of the function, to determine the value - the result of the function, use the **return** statement.

Follow these instructions first; leads to the end of execution of the program code of the function, and secondly, the value specified after the **return** statement is returned by the function as a result.

**Function without arguments**

**Example 1**

the **my\_func ()** function takes no arguments and prints the word spam three times

*def my\_func ():*

*print (“spam”)*

*print (“spam”)*

*print (“spam”)*

*my\_func ()*

The function is first defined, then called. Instructions inside a function are executed only when it is called.

**Function with arguments**

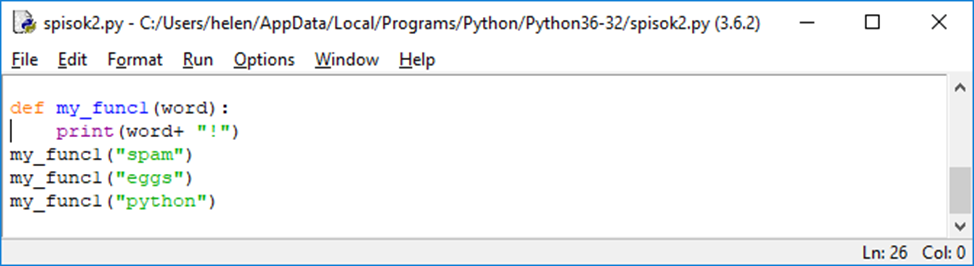
*def my\_func1(word):*

*print(word+“!”)*

*my\_func1(“spam”)*

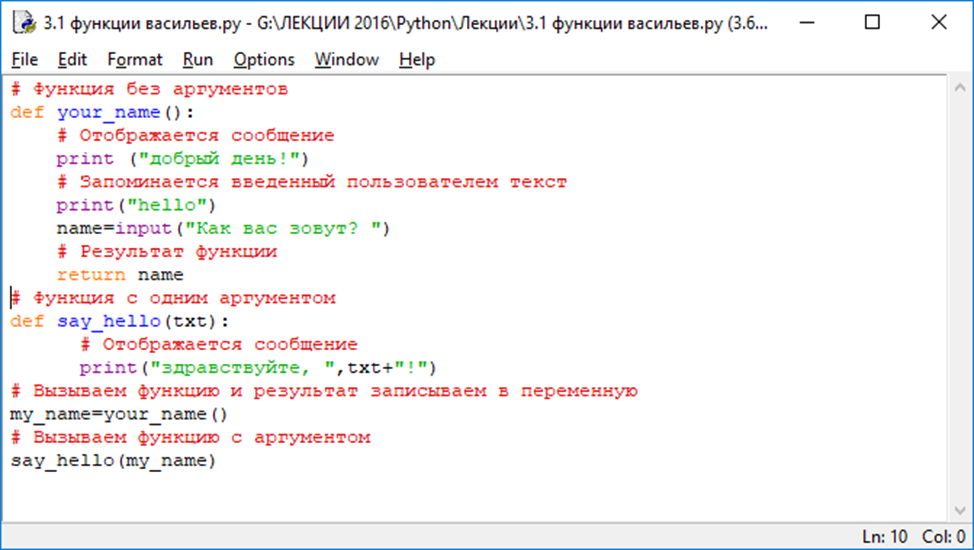
*my\_func1(“eggs”)*

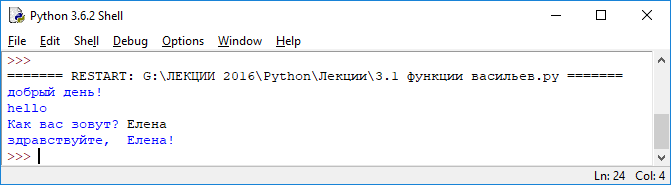
*my\_func1(“python”)*

****

**Example 2.**

Username output program - a simple example in which several simple functions are declared.





В программе объявляется две функции. Функция уоur\_nаmе ( ) не имеет аргументов. При выполнении этой функции сначала командой print ( " Добрый день ! " ) отображается приветствие. Затем пользователю предлагается ввести свое имя. Введенное пользователем текстовое значение запоминается в переменной name. Вся команда выглядит как name=input ( "Как Вас зовут ? " ) .

После этого с помощью инструкции return name значение переменной narne возвращается в качестве результата функции your\_name(). Таким образом, у этой функции нет аргументов, но зато она возвращает результат.

И ее результат - это то, что вводит пользователь (предполагается, что имя пользователя).

Еще одна функция say\_hello() нужна для отображения приветствия. У функции один аргумент (обозначен как txt). Текст сообщения, которое отображается при вызове функции, формируется с учетом значения аргумента txt, переданного функции. Результат функция не возвращает. В теле функции всего одна команда print (" Здравствуйте, " , txt + " ! " ) , которой в консольное окно выводится сообщение.

Описанные функции используем следующим образом. Сначала командой my\_name=your\_name() вызываем функцию your\_name( ) и результат вызова записываем в переменную my\_name. Затем командой say\_hello(my\_name) вызываем функцию say\_hello(), передав ей аргументом переменную my\_name. Результат этих действий таков, как показано выше.

***Task 1***.

Create a function to calculate f(x)= 2x+4x.

Calculate function values for x=2; 8; 11

def f(x):

x = 2\*\*x+4\*x

return x

print (f(2))

print(f(8))

print(f(11))

*Результат*

*12*

*288*

*2092*

**2. Scope (area of visibility) of variables: Global and local variables**

After the instructions are executed in the body of the function, the program transfers control to the point that follows the call of this function.

This ***principle of modularity***, which isolates parts of a program that must be run periodically, is very useful in programming.

When creating custom functions, you need to understand the principle of accessibility of variables in a program ***- the scope of variables***.

• Variables created outside a function can be accessed from instructions inside functions - they are **global**.

• Variables created inside functions cannot be accessed from outside - they have a local area. The limited availability of local variables means that variables with the same name can appear in various functions without any consequences.

**A variable is local** (visible only inside a function) if it is assigned a value inside functions

**A variable is global** if it is visible (you can access it) in the entire program, including inside the function.

If you want a local variable to be accessed from anywhere, you must first declare it using the global keyword followed by the variable name.

After that, you can assign a value to it as many times as you like, and it will be available from anywhere in the program.

In cases where two variables - global and local - have the same name, the function will use the local version.

Variables that are not global, but occur in some outer scopes, can be used by declaring them with the nonlocal keyword.

**Example 3**.

файл с именем myprog.py:

*a = 3 # глобальная переменная*

*print('глобальная переменная a = ', a)*

*y = 8 # глобальная переменная*

*print ('глобальная переменная y = ', y)*

*def func ():*

*print ('func: глобальная переменная a = ', a)*

*y = 5 # локальная переменная*

*print ('func: локальная переменная y = ', y)*

*func() # вызываем функцию func()*

*print('??? y = ', y) # отобразится глобальная переменная*

======= RESTART: C:/Python35-32/myprog.py =========

глобальная переменная a = 3

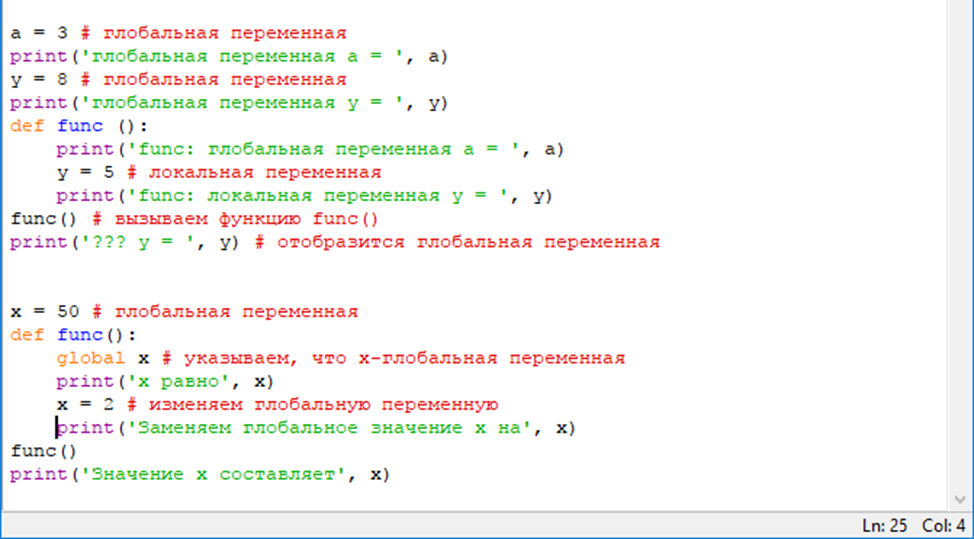
глобальная переменная y = 8

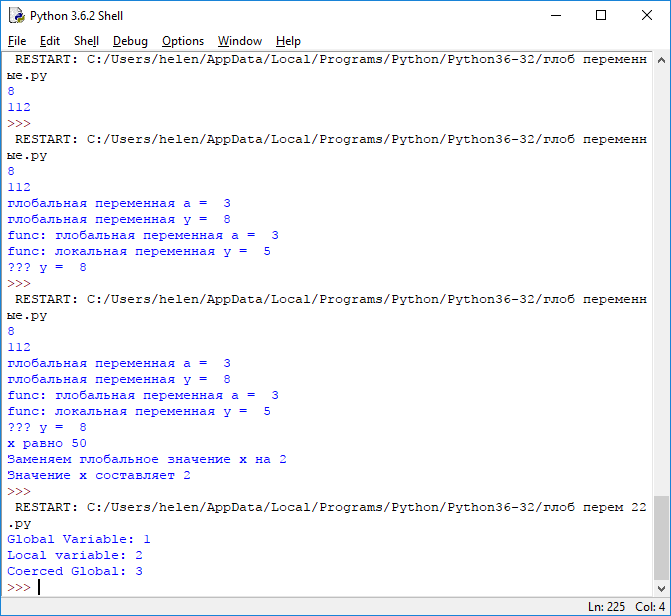
func: глобальная переменная a = 3

func: локальная переменная y = 5

??? y = 8

>>>

****



Inside the function, we were able to access the global variable a and display its value on the screen.

Next, a local variable y is created inside the function, and its name coincides with the name of the global variable - in this case, when accessing y, the contents of the local variable are displayed, and the global remains unchanged.

What if we want to change the contents of a global variable inside a function?

The following is an example of such a change using the **global** keyword.

**Example 3\_1**

*x = 50 # глобальная переменная*

*def func():*

*global x # указываем, что x-глобальная переменная*

*print ('x равно', x)*

*x = 2 # изменяем глобальную переменную*

*print ('Заменяем глобальное значение x на', x)*

*func()*

*print('Значение x составляет', x)*

**Useful features when working with functions in Python.**

1) Function names in Python are variables that contain the address of an object of type function, so this ***address can be assigned to another variable and a function with a different name can be called***.

def summa(x, y):

return x + y

f = summa

v = f(10, 3) # вызываем функцию с другим именем

2) Function parameters can take default values (значения по умолчанию):

def summa(x, y=2):

return x + y

a = summa(3) # вместо y подставляется значение по умолчанию

b = summa(10, 40) # теперь значение второго параметра равно 40

3) The function name is an ordinary variable, so we can pass it as an argument when calling the function:

def summa(x, y):

return x + y

def func(f, a, b):

return f(a, b)

v = func(summa, 10, 3) # передаем summa в качестве аргумента

**Example 4.**

**How from the func () function you can call the summa () function.**

In addition, at the time of calling the function, you can assign values to specific parameters (use key arguments):

def func(a, b=5, c=10):

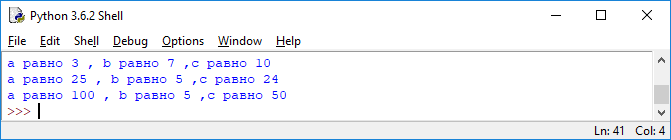
print('a равно', a, ', b равно', b, ',c равно', c)

func(3, 7) # a=3, b=7, c=10

func(25, c=24) # a=25, b=5, c=24

func(c=50, a=100) # a=100, b=5, c=50

An error will be a function call in which the **a** argument is not specified, since no default value is specified for it.



**3 Substitution and passing arguments**

When defining a user-defined function in Python, you can also specify an optional parameter (argument). After that, you can pass a value to the argument, indicating it in brackets when calling the function, and then it will use the value passed to it, referring to the name of the argument.

A function definition may have several parameters, and it may turn out that several arguments must be passed when the function is called.

There are several ways to pass arguments to functions (***способов передачи аргументов функциям***)

* Positional arguments (**Позиционные аргументы)** are listed in the order exactly corresponding to the order in which the parameters are written;
* Named arguments (**Именованные аргументы)** consist of a variable name and a value; Finally, there are lists and dictionaries of meanings.

**3. 1 Positional arguments**

When calling a function, each argument must be assigned a parameter in the function definition.

The easiest way to do this is based *on the order in which the arguments are listed.*

The values associated with the arguments in this way are called *positional arguments*.

If you need to define a function with more than one argument, they must be separated by commas.

***Example 5.* Positional arguments**

def my\_func2(x,y):

print(x +y)

print(x +y)

my\_func2 (5, 8)

*Результат:*

13

13

***Example 6.*** Функция для вывода информа­ции о домашних животных. Функция сообщает тип животного и его имя:

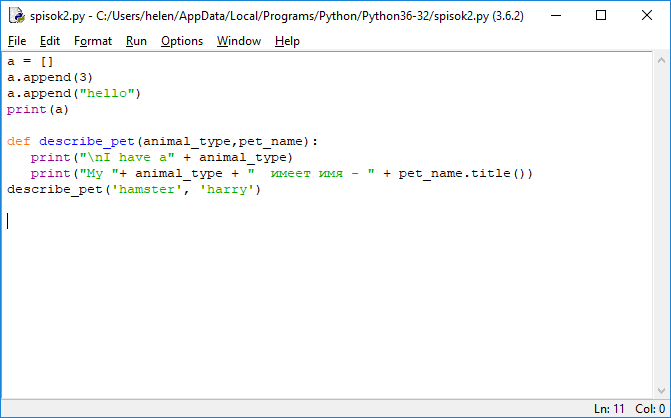
*pets.py*

def describe\_pet(animal\_type,pet\_name):

print("\nI have a" + animal\_type)

print("My "+ animal\_type + " имеет имя - " + pet\_name.title())

describe\_pet('hamster', 'harry')



**Task 2.**

Define a function that takes two arguments, multiplies them and prints the result.

*def my\_fun3(x,y):*

*print(x\*y)*

*my\_func3 (5, 8)*

**Parameters** are variables in the body of the function,

**arguments** - values assigned to parameters when the function is called

**Task 3***.*

***Define a function even, which will output “Yes” if its parameter is an even number, otherwise “No”***

*def even(x):*

*if x%2= = 0:*

*print(“Yes”)*

*else:*

*print(“No”)*

*even (5)*

**3.2. Multiple Function Calls**

The function can be called in the program as many times as needed.

***Example 7.***

def my\_func2(x,y):

print(x +y)

print(x +y)

my\_func2 (5, 8)

my\_func2 (1, 2)

*Результат:*

13

3

Для вывода информации о другом животном достаточно одного вызова describe\_pet() в Примере 5:

def describe\_pet(animal\_type,pet\_name):

print("\nI have a " + animal\_type)

print("My "+ animal\_type + " имеет имя - " + pet\_name.title())

describe\_pet('hamster', 'harry')

describe\_pet(‘dog’, ‘willie’)

Во втором вызове функции describe\_pet() передаются аргументы 'dog' и 'willie'. По аналогии с предыдущей парой аргументов Python сопоставляет аргумент 'dog' с параметром animal\_type, а аргумент 'willie' с параметром pet\_name.

Как и в предыдущем случае, функция выполняет свою задачу, но на этот раз вы­водятся другие значения:

I have a hamster

My hamster имеет имя - Harry

I have a dog

My dog имеет имя – Willie

Многократный вызов функции — чрезвычайно эффективный способ работы. Код вывода информации о домашнем животном пишется один раз в функции

**3.3. Named Arguments**

A named argument is a “***name-value***” pair passed to a function.

The *name* and the *value* are directly associated with the argument, so that when the argument is passed, confusion is excluded from the order.

Named arguments eliminate the hassle of ordering arguments when calling a function, and also clarify the role of each value in a function call.

Rewrite the program *pets.py* using named arguments when calling

describe\_pet ():

def describe\_pet (animal\_type,pet\_name):

print ("\nI have a " + animal\_type)

print ("My "+ animal\_type + " имеет имя - " + pet\_name.title())

describe\_pet (animal\_type='hamster', pet\_name='harry')

ИЛИ

describe\_pet (pet\_name='harry', animal\_type='hamster')

Функция describe\_pet() не изменилась. Однако на этот раз при вызове функ­ции мы явно сообщаем Python, с каким параметром должен быть связан каждый аргумент. При обработке вызова функции Python знает, что аргумент 'hamster' должен быть сохранен в параметре animal\_type, а аргумент 'harry' в параметре pet\_name.

Порядок следования именованных аргументов в данном случае не важен, потому что Python знает, где должно храниться каждое значение. Следующие два вызова функции эквивалентны:

describe\_pet(animal\_type='hamster', pet\_name='harry')

describe\_pet(pet\_name='harry', animal\_type='hamster')

**4. Return values - return**

A ***user function*** (***Пользовательская функция***) can also return values to the operator that called it.

This is done using the return keyword followed by the return value.

**Example 8.** Function to return the value of the sum of two folding arguments

***def sum (a , b ):***

***return a + b***

The returned result can be assigned to a variable using the function call instruction and subsequently used in the program, for example:

***total = sum( 8 , 4 )***

***print(total )***

Либо он может быть использован *непосредственно «на лету»:*

***Print (sum( 8 , 4 ) )***

**Example 9**

*def max(x, y):*

*if x>=y:*

*return x*

*else:*

*return y*

*print (max(4, 7))*

*z=max(8, 5)*

*print(z)*

*результат*

*7*

*8*

**Task 4.**

Determine the *shorttest* function, which compares the length of its arguments and returns the shortest

*def shortest (x, y):*

*if len(x)<= len(y):*

*return x*

*else:*

*return y*

*print (shortest ("abc","adcdg"))*

результат

abc

After the function returns a value, it is no longer executed.

Whatever code goes after **return**, it will not be executed.

*def shortest (x, y):*

*if len(x)<= len(y):*

*return x*

*print (“Х= ”, x) # Не выполняется*

*else:*

*return y*

*print (shortest ("abc","adcdg"))*

**Task 5.** What will the program return?

*def shout (word):*

*return word+”!”*

*speak= shout*

*output= speak(“shout”)*

*print (output)*

*результат*

1. **shout!**
2. speak!
3. word!

Functions can also be used as arguments for other functions.

**Example 10**

*def add (x, y):*

*return x+y*

*def do\_twice (func, x, y):*

*return func (func(x, y), func(x, y))*

*a=5*

*b=10*

*print (do\_twice(add, a, b))*

*Результат*

*30*

***The Basic literature***

1. Мэтиз Эрик.Изучаем Python. Программирование игр, визуализация данных, веб-приложения. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Библиотека программиста»).

2. [Mark Lutz](http://www.allitebooks.org/author/mark-lutz/). Learning Python. – O’Reilly, 2013.

3. Allen B. Downey. Think Python. - O’Reilly, 2015.

4. Brett Slatkin.  Effective Python: 59 Specific Ways to Write Better Python. - 2015. – 256 p.

5. [Mark Lutz. Programming Python: Powerful Object-Oriented Programming](https://www.amazon.com/Programming-Python-Powerful-Object-Oriented/dp/0596158106/ref=dp_ob_title_bk) O'Reilly Media, 2016. – 1552 p.

6. МакГрат Майк. Программирование на Python для начинающих. – М.: Эксмо, 2015. -  192 с.

***Additional***

7. Саммерфилд М. Программирование на Python 3. Подробное руководство. – Пер. с англ. – СПб.: Символ\_Плюс, 2009. – 608 с., ил.

8. Златопольский Д. М. Основы программирования на языке Python. – М.: ДМК Пресс, 2017. – 284 с.: ил.

***Internet resources***

http://www.python.org/

http://pythontutor.ru/

http://www.enthought.com/products/epdlibraries.php

http://www.edu.ru/modules.php?op=modload&name=Web\_Links&file=index&l\_op=viewlink

https://docs.python.org/3/tutorial/index.html

http://www.python.org/

http://pythontutor.ru/

**Theme 7. Functional programming**

1 **lambda** functions

2. Built-in (встроенные) functions **map** and **filter**

3 Generators in Python

4. Module **itertools**

**Functional programming**

**Functional programming** is a programming paradigm that is based on functions.

One of the key concepts of functional programming is a **higher order function (функции высшего порядка)**.

**Higher-order functions** take other functions as arguments, or return them as a result.

**Task 1.** Determine the result of the program

def apply\_twice (func, arg):

return func(func(arg))

def add\_five (x):

return x+5

print (apply\_twice(add\_five, 10))

*Результат*

*20*

Функция apply\_twice принимает другую функцию в качестве аргумента и вызывает ее дважды внутри своего тела

**1. lambda-functions**

You can use the def keyword to create a named function and then use its name to call anywhere in your program. In addition, a named function can return some value to the operator that called it.

When a function (def) is created, it is bound to a variable automatically.

Other objects, such as strings and integers, are created on the fly without assigning variables to them.

The same thing happens with a function if it is created using a lambda function. Such functions are known as ***anonymous functions***.

**lambda-functions - are created in the course of work, without assigning variables to them - anonymous functions (анонимные функции)**

**Task 2**

*def my\_func7 (f, arg):*

*return f(arg)*

*или*

*my\_func (lambda x: 2\*x\*x, 5)*

**lambda functions are not as functional as named functions.**

**Their functionality is expressed in one expression.**

**Task 3**

#named function

*def my\_func8 (x):*

*return x\*\*2+5\*x+4*

*print (my\_func8 (-4))*

*# lambda function*

*print ((****lambda x: x\*\*2+5\*x+4)(-4))***

*результат*

*0*

*0*

Lambda functions can be assigned variables and they can be used as normal functions.

**double = lambda x: x \* 2**

print (**double** (7))

*результат*

*14*

**Task 3.** Determine the function of raising the number 5 to the power 2 in two ways

**def square( x ) :**

**return x \*\* 2**

print (**square** (5))

можно в более лаконичной форме записать так:

**square = lambda x : x \*\* 2**

print (**square** (5))

В обоих случаях вызов **square(5)** возвратит результат 25, передав целое число 5 в качестве аргумента функции. Отметим, что аргумент после ключевого слова **lambda** стоит без скобок и выражение в данной функции не требует ключевого слова **return**, поскольку **lambda**-функция в любом случае возвращает значение.

**Task 4.** Determine the result of the program

**triple = lambda x : x \*3**

add**= lambda x, y : x + y**

print (**add (triple** (3), 4)

*результат*

*13*

**2. Map and filter functions**

The built-in map and filter functions are higher-order functions for working with lists (or with objects to be iterated).

The map function takes a function and an object to be iterated as its arguments and returns a new object to be iterated, and the function is applied to each argument.

**Example**

def add\_five (x):

return x+5

nums = [11,22,33,44,55]

result = **list (map(add\_five, nums))**

print (result)

*Результат:*

[16,27,38,49,60]

Можно добиться того же результата используя функцию **lambda**

nums = [11,22,33,44,55]

result = list (map(**lambda x: x+5, nums**))

print (**result**)

чтобы преобразовать результат в список, использовали функцию **list**

**Task 5.** Fill in the blanks in the program to multiply each item in the list by 2 using the lambda function

nums = [11,22,33]

a = list (map(**\_\_\_\_\_\_\_\_\_\_\_: \_\_\_\_\_, \_\_\_\_\_**))

print (**a**)

*результат*

nums = [11,22,33]

a = list (map(**lambda x: x\*2, nums**))

print (**a**)

The **filter** function is designed to filter the iterable object by removing elements that do not match the predicate (a function that returns a boolean)

**Example**

nums = [11,22,33,44,55]

res = list (filter (**lambda x: x%2 = = 0, nums**))

print (**res**)

*результат*

[22, 44]

**Task 6.** Fill in the blanks in the program to remove from the list all items greater than 4

nums = [1,2,5,8,3,0,7]

res = list (map(**lambda x: \_\_\_\_\_, \_\_\_\_\_**))

print (**res**)

*результат*

nums = [1,2,5,8,3,0,7]

res = list (filter (**lambda x: x<5, nums**))

print (**res**)

**3. Generators in Python - yield**

In Python, there is a special function generator that returns an object, not a value.

At the same time, it saves the state of its last call and on the next call continues to work from the same point.

Function-generators look just like regular ones, but they contain expressions with the **yield** keyword for successively generating values that can be used in **for in** loops, or getting them using the **next ()** function.

At each yield, the work of the function is temporarily suspended, while the execution state is maintained, including local variables, a pointer to the current instruction, an internal stack, and an exception handling state.

Upon subsequent access to the generator iterator (when calling its methods), the function continues its execution from the place where it was suspended.

This function generators are different from the usual functions, when you call which the execution starts from the beginning every time.

If the function reaches the *return instruction*, or the end (without specifying the mentioned instruction), an exception is raised and the iterator exhausts itself.

Generators are created using functions and **yield** instructions.

**Example**

def countdown ():

i=5

while i>0:

yield i

i -= 1

for i in countdown ():

print(i)

*Результат:*

5

4

3

2

1

Инструкции **yield** определяет генератор, заменяет значение возвращаемое функцией, и возвращает результат не меняя первоначальные переменные

**Since the generators return one element at a time, they, unlike lists, have no memory limitations. They can run infinitely.**

**Example**

def func\_sevens ():

while True:

yield 7

for i in func\_sevens ():

print(i)

*Результат:*

7

7

7

7

…

Таким образом, генераторы позволяют объявить функцию, которая подобна итератору, т.е. может быть использована в цикле

**The final generators can be converted into lists, for this they need to be passed as arguments to the list function.**

*Example 11*

def numbers (x):

for i in range(x):

if i%2 = = 0:

yield 7

print (list (numbers(11)))

*Результат:*

[0,2,4,6,8,10]

**Generators in Python - yield**

Using generators improves performance: “lazy” generation of values (generation on demand) means lower memory consumption.

In addition, you do not need to wait until all the elements are generated, we can start using them immediately.

**Task 7.** Determine the result of the program

def make\_word ():

word = “”

for ch in ‘spam’:

word += ch

yield word

print (list (make\_word()))

*Результат*

1. [‘spam’, ‘spam’, ‘spam’, ‘spam’]
2. ‘spam’
3. **[‘s’, ‘sp’, ‘spa’, ‘spam’]**

**The object returned by the yield statement can be assigned to a variable**

def incrementer() :

i = 1

while True :

yield i

i += 1

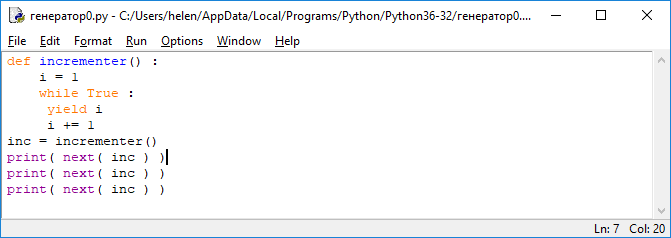
inc = incrementer()

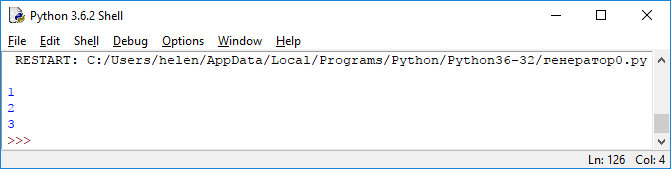
print( next( inc ) )

print( next( inc ) )

print( next( inc ) )

Эти последовательные вызовы функции выведут целые значения **1**, **2** и **3**





**4 The itertools module**

**The itertools module** is a standard library that contains several useful functions in functional programming.

One type of function in this library is infinite iterators (бесконечные итераторы)**.**

**The count function** creates an infinite progression up from a given number.

**Function cycle** - an infinite number of times iterates through the object to be iterated (for example, a string or a list).

**The function repeat** -repeats an object infinitely or a specified number of times.

*Example*

from itertools import count

for i in count(3):

print (i)

if i>= 11:

break

*Результат*

3

4

5

6

7

8

9

10

11

**Task**

Fill in the blanks to import the cycle function from the **itertools module:**

\_\_\_\_\_\_\_itertools \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

from itertools import **cycle**

**The itertools library also has functions for working with iterable objects:**

**takewhile -** returns elements from the object being iterated that satisfy the predicative function

**chain -** combines several iterable objects into one

**accumulate -** returns the sum of values within the object being iterated

*Example***:**

from itertools import **accumulate**, **takewhile**

nums = list (**accumulate (range(8)))**

print (nums)

print (list (**takewhile (lambda x: x <= 6, nums)**))

*Результат:*

[0,1,3,6,10,15,21,28] 0+1, 0+1+2, 0+1+2+3, 0+1+2+3+4……

[0,1,3,6]

**Task**

Fill in the blanks so. For the program to return even numbers from the list, using the **takewhile function**

from itertools import **\_\_\_\_\_\_\_\_\_\_\_\_**

nums = [2,4,6,7,9,8]

**a= \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_( \_\_\_\_\_\_\_\_\_\_ x: x%2 = = 0, nums)**

**print (list (a))**

*Результат:*

from itertools import **takewhile**

nums = [2,4,6,7,9,8]

**a= takewhile (lambda x: x%2 = = 0, nums)**

**print (list (a))**

***The Basic literature***

1. Мэтиз Эрик.Изучаем Python. Программирование игр, визуализация данных, веб-приложения. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Библиотека программиста»).

2. [Mark Lutz](http://www.allitebooks.org/author/mark-lutz/). Learning Python. – O’Reilly, 2013.

3. Allen B. Downey. Think Python. - O’Reilly, 2015.

4. Brett Slatkin.  Effective Python: 59 Specific Ways to Write Better Python. - 2015. – 256 p.

5. [Mark Lutz. Programming Python: Powerful Object-Oriented Programming](https://www.amazon.com/Programming-Python-Powerful-Object-Oriented/dp/0596158106/ref=dp_ob_title_bk) O'Reilly Media, 2016. – 1552 p.

6. МакГрат Майк. Программирование на Python для начинающих. – М.: Эксмо, 2015. -  192 с.

***Additional***

7. Саммерфилд М. Программирование на Python 3. Подробное руководство. – Пер. с англ. – СПб.: Символ\_Плюс, 2009. – 608 с., ил.

8. Златопольский Д. М. Основы программирования на языке Python. – М.: ДМК Пресс, 2017. – 284 с.: ил.

***Internet resources***

http://www.python.org/

http://pythontutor.ru/

http://www.enthought.com/products/epdlibraries.php

http://www.edu.ru/modules.php?op=modload&name=Web\_Links&file=index&l\_op=viewlink

https://docs.python.org/3/tutorial/index.html

http://www.python.org/

http://pythontutor.ru/

**Theme 8 Types of collections: sets, dictionaries, tuples**

1. Tuples

2. Sets

3. Dictionaries

**1. Tuples**

**Lists** are well suited for storing sets of items that may change throughout the program's life cycle.

An **immutable list** (Неизменяемый список) in Python is called a **tuple (кортеж)** and is created by assigning values separated by commas and placed in parentheses.

The tuple data type can be called as a **tuple ()** function — it returns an empty tuple without arguments.

The tuple data type can be called as a **tuple ()** function - without arguments, it returns an empty tuple, with an argument of type tuple returns a shallow copy of the argument; in case the argument has a different type, an attempt is made to convert it to an object of type tuple.

An **empty tuple** is created using a pair of empty parentheses ().

**a = (1, 2, 5, 50)**

**Tuple definition**

*pr = (200, 50)*

After defining a tuple, it is possible to refer to its individual elements by indices in the same way as this is done when working with a list.

*Example.*

A rectangle must always have strictly defined dimensions.

To ensure that sizes do not change, you can combine dimensions into a tuple:

*pr = (200, 50)*

*print(pr[0])*

*print(pr[1])*

When trying to change one of the elements in the *pr* tuple:

***pr = (200, 50)***

***pr[0] = 250***

Python returns an error like:

*TypeError: 'tuple' object does not support item assignment*

**Items in a tuple**

In Figure 1 shows the tuple t = "venus", –28, "green", "21", 19.74 and the indices of the elements inside the tuple.

****

Figure 1

Lines are indexed in exactly the same way, but if there is a single character in the lines of each position, then in the tuples of each position there is a single object reference.

**Iterate over all values in a tuple**

To iterate over all values in a tuple, a for loop is used, as with lists:

*pr = (200, 50)*

*for i in pr:*

*print (i)*

Python returns all the elements of a tuple, similar to how this is done with a list:

200

50

**Replacing a tuple**

Elements of a tuple cannot be changed, but you can assign a new value to the variable in which the tuple is stored. Thus, to resize the rectangle, the whole tuple should be redefined:

*pr = (200, 50)*

*print("Original dimensions:")*

*for i in pr:*

*print(i)*

*pr = (400, 100)*

*print("\nModified dimensions:")*

*for i in pr:*

*print(i)*

The values ​​of the elements of a regular list can be changed by the programmer as the program executes.

But a list can also be created with fixed (immutable) values ​​that remain constant throughout the program.

An immutable list in Python is called a tuple and is created by assigning values ​​separated by commas and in parentheses.

This assignment process is called **tuple packing**:

**k = ('Red', 'Green', 'Red', 'Blue', 'Red')**

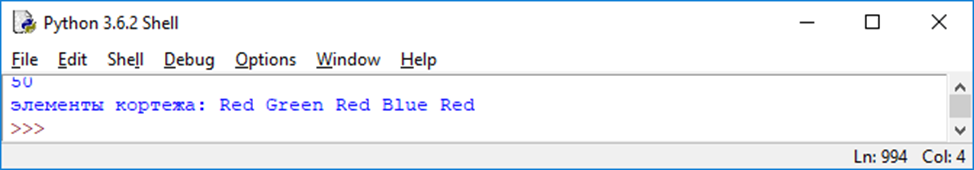
A single tuple element can be accessed using the tuple name and index entry in square brackets. All values ​​stored inside a tuple can be assigned to separate variables. This process is called **tuple unpacking**:

**b,c,d,e,f = k**

**print(b,c,d,e,f)**

Like lists, tuple elements are indexed starting at zero.

**To unpack a tuple**, the number of variables must be equal to the number of elements in the tuple

****

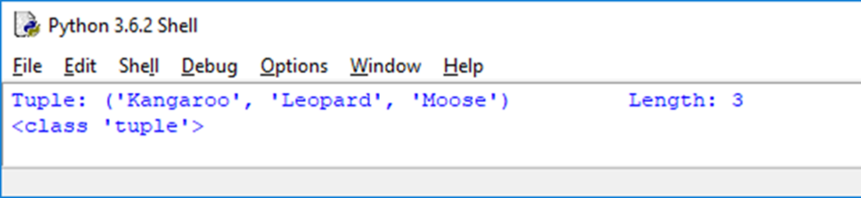
**Program.**

Initialize the tuple, then output its contents, length, and type.

**zoo = ( 'Kangaroo' , 'Leopard' , 'Moose' )**

**print( 'Tuple:' , zoo , '\tLength:' , len( zoo ) )**

**print( type( zoo ) )**

****

**2. Sets (множества)**

**Python set (Множество)** is an unordered collection of immutable, unique elements

**v = {'A', 'C', 4, '5', 'B'}**



Python has two built-in types of sets:

* a changeable type **set**  (изменяемый тип set)
* a fixed type **frozenset** (фиксированное - frozenset.)

When iterating over elements of a set, elements may follow in any order.

**An immutable list** of unique values is called a **set** and is created

*phonetic-set = {'Alpha', 'Bravo', 'Charlie'}*

**Ways to create a set**

v = {'A', 'C', 4, '5', 'B', 4}

print (‘v=‘, v)

v= {'C', 'B', '5', 4, 'A'}

Duplicate elements that were added when creating a set were deleted (the elements of the set are unique)

v = set ([3,6,3,5])

print (‘v=‘, v)

v={3, 5, 6}

The **range ()** function allows you to create sets from a range:

v = set (range (10))

v={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

**Ways to create a set**

Sets can be created from lists.

At the time of creating the set from the list, duplicate elements will be deleted.

This is a way to clear the list of repeats:

**s2 = list (set ([7,3,6,3,7,5]))**

**print(s2)**

**[3, 5, 6, 7]**

*You cannot refer to an individual element of a set using the name of the set and square brackets containing the index*.

Instead, sets have methods for working with them.

**Метод множества Описание**

**set.add(*x*)** Добавляет элемент ***х*** в множество

**set.update(*x*,*y*,*z*)** Добавляет несколько элементов в множество

**set.copy()** Возвращает копию множества

**set.pop()** Удаляет один элемент из множества случайным образом

**set.discard( *i* )** Удаляет из множества элемент с порядковым номером ***i***

**set1.intersection(*set2*)** Возвращает элементы, принадлежащие **обоим**

set1 & set2 **множествам (пересечение)**

**set1.difference(*set2*)** Возвращает элемента из множества ***set1***, которых нет в

set1 – set2 ***set2 (*разность*)***

***set1***.**union**(***set2***) Возвращает новое множество, включающее все элементы

***set1*** | set2 множества **set1** и все элементы множества set2, отсутствующие

в множестве set1 **(объединение)**

***set1***.update(***set2***) Добавляет во множество **set1** все элементы множества **set2**,

set1 |= set2 отсутствующие в множестве **set1**

**Operations on sets**

>>> s1=set(range(5))

>>> s2=set(range(2))

>>> s1

{0, 1, 2, 3, 4}

>>> s2

{0, 1}

>>> s1.add('5') # добавить элемент

>>> s1

{0, 1, 2, 3, 4, '5'}



>>> s1.intersection(s2) # пересечение множеств через вызов метода (s1 & s2)

{0, 1}

>>> s1.union(s2) # объединение множеств через вызов метода (s1 & s2)

{0, 1, 2, 3, 4, '5'}

>>>

**Set operations:**

**len () -** finding the length

**in and not in** - quick check for element occurrence

**Example.**

Initialize the set, then add another element to it and then output its contents, length and type.

*bag = { 'Red' , 'Green' , 'Blue' }*

*bag.add ( 'Yellow' )*

*print ( '****\n****Set:' , bag , '****\t****Length' , len( bag ) )*

*print ( type( bag ) )*

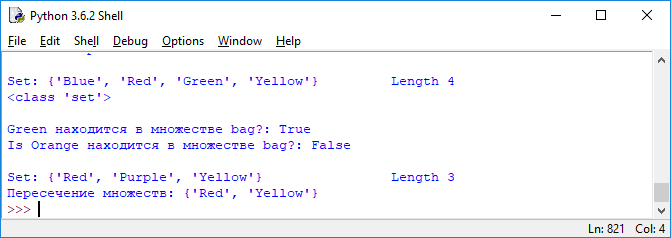
*print( ‘****\n****Green находится в множестве bag?:' , 'Green' in bag )*

*print( 'Is Orange находится в множестве bag?:' , 'Orange' in bag )*

*box = { 'Red' ,'Purple' , 'Yellow' }*

*print( '****\n****Set:' , box , '****\t****Length' , len( box ) )*

*print( 'Пересечение множеств:' , bag.intersection( box ) )*



**3. The dictionary**

The **dictionary** (dict) in Python is an unordered mutable collection or, a “list” with arbitrary keys of unchangeable type.

**The dictionary is a container that can contain several data elements as a set of pairs** ***key: value***.

Values in the dictionary can be ***accessed by the key associated with it*.**

The key must be unique within this dictionary and is usually a string, although numeric values are also sometimes used.

**Creating a dictionary** is an assignment to a variable with a structure name in curly brackets that you selected, in which the string pairs ‘key: value’ are separated by commas.

Strings must be enclosed in quotation marks, and the colon must be between the key and the value associated with it.

**Pair key: value** can be deleted from the dictionary.

To do this, specify the dictionary name and the key name in square brackets after the del keyword.

Conversely, by assigning a value to a new key, you can add a key: value pair to an existing dictionary.

For dictionaries in Python, there is a **keys ()** method, with which you can return a list of all dictionary keys in a random order using the suffix entry for the dictionary name.

The **sorted ()** function is used to sort keys in alphanumeric order.

Using the **in** operator, you can determine whether the dictionary contains the desired key.

To do this, use the syntax key **in the dictionary**.

The search will return a boolean value **True** if the key is found in the specified dictionary; otherwise, the value False.

**Ways to create a dictionary**

*d1 = dict({"id": 1948, "name": "Washer", "size": 3})*

*# Словарь d1 создается с помощью литерала словаря.*

*d2 = dict(id=1948, name="Washer", size=3)*

*# Словарь d2 создается с помощью именованных аргументов.*

*d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])*

*d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3)))*

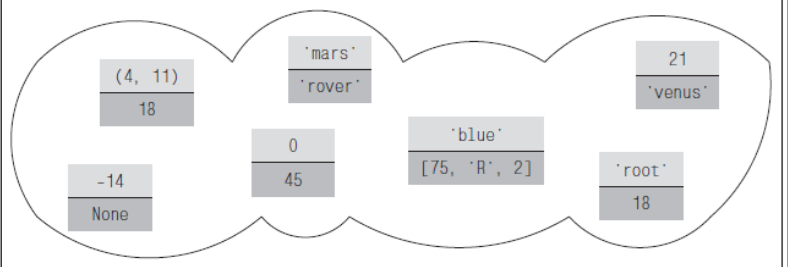
*d5 = {"id": 1948, "name": "Washer", "size": 3}*

*# Словари d3 и d4 создаются из последовательностей, а словарь d5 создается из литерала словаря.*

*print(d1,'\n',d2,'\n',d3,'\n',d4,'\n',d5)*

*{'id': 1948, 'name': 'Washer', 'size': 3}*

**d = {"root": 18, "blue": [75, "R", 2], 21: "venus", -14: None, "mars": "rover", (4, 11): 18, 0: 45}**

****

To access the individual elements are used *square brackets* (квадратные скобки):

for example, *the expression d ["root"] returns 18,*

*d [21] returns the string "venus"*,

and the expression d [91] in relation to the dictionary shown in Fig. 2, throws a KeyError exception.

Dictionaries support the built-in **len ()** function.

Using the in operator, you can determine whether the dictionary contains the desired key.

***key* in *dictionary.***

*For example:*

**print ("root" in d)**

The search will return the boolean value True if the key is found in the specified dictionary; otherwise, False appears.

**Dictionary methods**

**d.items()** Возвращает представление всех пар (ключ, значение) в словаре d

**d.keys()**  Возвращает представление всех ключей словаря d

**d.values()** Возвращает представление всех значений в словаре d

**d.get(k)** Возвращает значение ключа k или None, если ключ k отсутствует в словаре

**d.get(k, v)** Возвращает значение ключа k или *v*, если ключ k отсутствует в словаре

**d.pop(k)** Возвращает значение ключа k и удаляет из словаря элемент с ключом k или возбуждает исключение KeyError, если ключ k отсутствует в словаре

**d.pop(k, v)** Возвращает значение ключа k и удаляет из словаря элемент с ключом k или возвращает значение *v*, если ключ k отсутствует в словаре

**d.update (a)** - Добавляет в словарь d пары (ключ, значение) из a, которые отсутствуют в словаре d, а для каждого ключа, который уже присутствует в словаре d, выполняется замена соответствующим значением из a; a может быть словарем, итерируемым объектом с парами (ключ, значение) или именованными аргументами

**Bypassing *(обход)* the values in the dictionary:**



*d = {"root": 18, "blue": [75, "R", 2], 21: "venus", -14: None, "mars": "rover", (4, 11): 18, 0: 45}*

*for i in d.values():*

*print (i)*

18

[75, 'R', 2]

venus

None

rover

18

45

If you need to change the values in the dictionary, you can loop through the dictionary keys and change the values using the square brackets operator.

An example of how you can increment all values in a dictionary d, assuming that all values are numbers:

**for key in d:**

**d[key] += 1**

The **dict.items (), dict.keys (), and dict.values ()** methods return dictionary representations.

A dictionary view is actually a read-only iterable object that stores the elements, keys, or values of the dictionary depending on which view was requested.

**Theme 9. The Basics of Object Oriented Programming in Python**

1. Data Encapsulation

2. Creating Object Instances

3. Access to class attributes

4. Inline Attributes

5. Garbage collection

6. Inheritance of properties

7. Redefining Basic Methods

8. Implementation of polymorphism

**Object oriented programming**

The three cornerstones of object-oriented programming (OOP) are:

1. encapsulation (**инкапсуляция)**
2. inheritance (**наследование)**
3. polymorphism (**полиморфизм)**

**1. Data Encapsulation**

A class is a type that describes a set of properties that characterize an object.

Each class has a data structure that can contain both functions and variables characterizing the object.

Members of a class can be functions called methods, as well as variables (declared within the structure of a class) called attributes.

Members of a class can be accessed in the program using a dot entry, putting the appropriate suffix after the class name

*Syntax*

**class-name.name-method () or class-name.atribute-name**

A class declaration (Объявление класса) begins with the class keyword, followed by the class name (when choosing a name, the usual Python naming rule follows, but begin with uppercase letters (букв в верхнем регистре)), then a colon (*символ двоеточия)* is used.

Next are indented instructions defining the class documentation string, the declaration of variables — class attributes, and the definition of class methods.

*Class block syntax:*

**Class ClassName:**

**'' 'class-documentation-string' ''**

**class variable declaration**

**class definition-methods**

**The declaration of a class,** in which its attributes and methods are defined, is a *sample* (образец) from which working copies (*instances* - экземпляры) of a class can be produced.

***All variables declared inside method definitions*** are known as instance variables (переменные экземпляра) and are only available locally in the method in which they were declared — they cannot be accessed from outside the class structure.

Data encapsulation ensures that data is securely stored in a class structure — *the first principle of object-oriented programming.*

**All properties of a class** can be accessed locally using a dotted entry **with the prefix self (**точечную запись с префиксом **self)**, for example, an attribute with the name *sound* can be called like this: *self.sound*.

All method definitions must contain **self** as the first argument, that is, a method called **talk** looks like **talk (self)**.

When creating an instance of a class, the special method **\_\_init \_\_ (self)** is automatically called.

If you need to pass more values to initialize its attributes, then subsequent arguments can be added to the brackets.

*Example* class declaration in Python

**class Critter :**

**''' Базовый класс для всех живых существ. '''**

**count = 0**

**def \_\_init\_\_( self , chat ) :**

**self.sound = chat**

**Critter.count += 1**

**def talk( self ) :**

**return self.sound**

Let's look at the **class components** presented in this example.

• The variable **count** is a class variable whose integer value is available to all instances of this class. It can be accessed using the Critter.count entry, both inside and outside the class.

• The first of the **\_\_init \_\_ ()** methods is the initialization method, which is called automatically when the class is instantiated.

• The **\_\_init \_\_ ()** method in this case initializes the sound instance variable with the value passed from the chat argument and increments the value of the class variable count each time an instance of this class is created.

• The second **talk ()** method is declared as a normal function, with the exception of the automatically specified self argument - no other values ​​are required for transmission from the called operator.

• In this case, the **talk ()** method returns the value encapsulated in the sound instance variable.

**2. Creating Object Instances (создание экземпляров объектов)**

An **instance of a class** that is an object is simply a copy of the prototype created by invoking a class constructor with the required number of arguments inside brackets when invoked.

  The arguments of this call must match those specified in the definition of the **\_\_init \_\_ ()** method, with the exception of the internal argument **self**.

**An instance of a class** is an object returned by the constructor, and can be assigned to a variable using the syntax

**instance\_name = NameClass (arguments)**

To access the methods and attributes of the created instance, you can use a dot entry:

**instance-name.name-method ()**

**or**

**instance\_name.atribute\_name**

**Task 1**

1. Create a Python program Bird.py - declare a new class containing a descriptive docstring.

class Bird:

'' 'The base class that defines the properties of birds.' ''

2. Add a statement declaring and initializing a variable - a class attribute with an integer value equal to zero.

count = 0

3. Define a class method that initializes an instance variable and increments the value of the class variable.

def \_\_init \_\_ (self, chat):

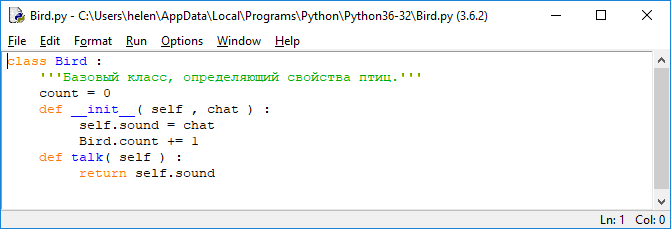
self.sound = chat

Bird.count + = 1

4. Add a class method that returns the value of an instance variable when called. Then save the class in a separate Bird.py file.

def talk (self):

return self.sound



5. Create a second Python program, make the class properties available by importing, and then print the docstring for the class.

from Bird import \*

print ('\ nClass Instances Of: \ n', Bird .\_\_ doc\_\_)

6. Add a statement to instantiate the class and pass the value of the string argument to the instance variable.

polly = Bird ('Squawk, squawk!')

7. Print the value of the instance variable and call the class method to display the value of the class variable.

print ('\ nNumber Of Birds:', polly.count)

print ('Polly Says:', polly.talk ())

8. Create a second instance of the class by passing a different value for the string argument to the instance variable.

harry = Bird ('Tweet, tweet!')

9. Output the value of the instance variable and call the class method to output the value of the class variable.

print ('\ nNumber Of Birds:', harry.count)

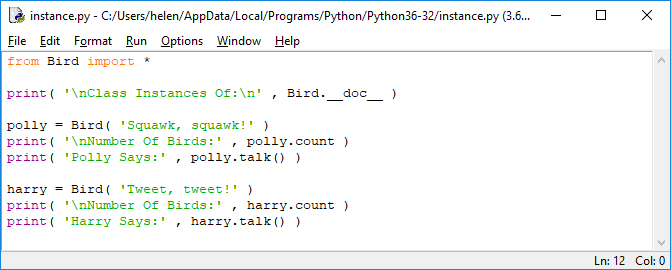
print ('Harry Says:', harry.talk ())

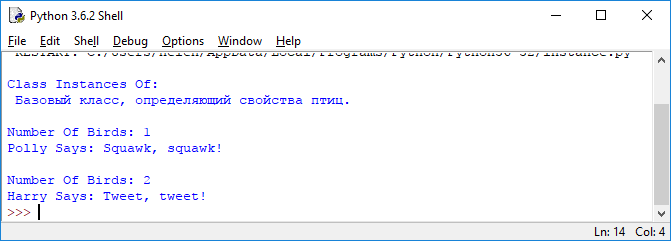
10. Save the instance.py file

Bird class instance - polly

Bird class instance - harry

The count class variable can also be accessed using the Bird.count entry, while the encapsulated sound instance variable can only be accessed by calling the talk () instance method.





**3. Access to class attributes**

Attributes of the created class instance can be added, changed or deleted at any time, using a dot entry to access them.

  If you build a statement in which you assign a value to an attribute, you can change the value contained within the existing attribute, or create a new one with the specified name and containing the assigned value:

***instance-name. attribute-name = value***

***del* *instance-name. attribute-name***

***built-in Python functions*:**

• **getattr( *имя-экземпляра* , *'имя-атрибута'* )** — возвращает значение атрибута экземпляра класса;

• **hasattr( *имя-экземпляра* , *'имя-атрибута'* )** — возвращает True, если значение атрибута существует в экземпляре, в противном случае возвращает False;

• **setattr( *имя-экземпляра , 'имя-атрибута'* , *значение* )** — модифицирует существующее значение атрибута либо создает новый атрибут для экземпляра;

• **delattr( *имя-экземпляра* , *'имя-атрибута'* )** — удаляет атрибут из экземпляра

**Task 2**

1. Create a Python program by making available the functions of the Bird class created in the previous example (Task 1).

from Bird import \*

2. Create an instance of the class and then add a new attribute with the assigned value using dot notation.

chick = Bird ('Cheep, cheep!')

chick.age = '1 week' \_\_

3. Output the values ​​stored in both attributes of the instance.

print ('\ nChick Says:', chick.talk ())

print ('Chick Age:', chick.age)

4. Modify the new attribute using dot notation and print its new value.

chick.age = '2 weeks'

print ('Chick Now:', chick.age)

5. Change the new attribute again, this time using the built-in function.

setattr (chick, 'age', '3 weeks')

Bird class instance - chick

6. List all of the instance's non-private attributes and their corresponding values ​​using the built-in function.

print ('\ nChick Attributes ...')

for attrib in dir (chick):

if attrib [0]! = '\_':

print (attrib, ':', getattr (chick, attrib))

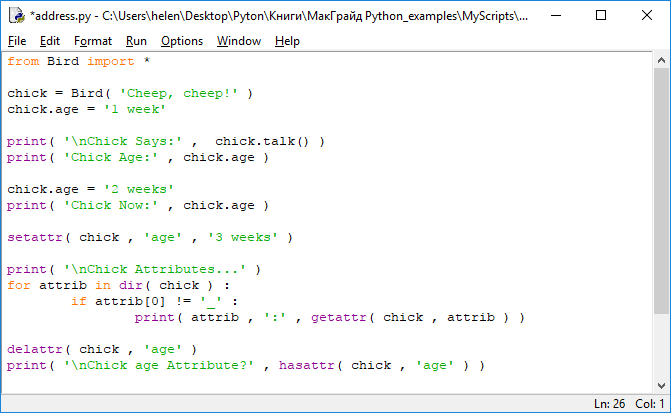
This loop skips all attributes whose names begin with an underscore, so that "private" attributes will not be included in the resulting list.

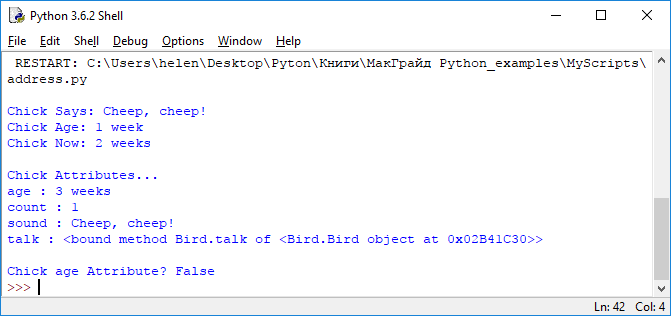
7. Remove the new attribute and check for its absence using the built-in functions.

delattr (chick, 'age')

print ('\ nChick age Attribute?', hasattr (chick, 'age'))

8. Save the file





**4. Built-in class attributes**

To get the value of the attribute string of the documentation of a particular class, you need to write the ***classname .\_\_ doc\_\_***

To display a list of all built-in attributes of a specified class, you can use the ***dir ()***function, which specifies the class name as a parameter, and then check whether the attribute name begins with an underscore.

The built-in attribute ***\_\_dict\_\_***is a dictionary containing pairs of keys and associated values. The keys here are the attribute names, and the values ​​are the corresponding attribute values.

The base class dictionary includes the ***\_\_init \_\_ ()***method, as well as all the methods and class attributes.

The class instance dictionary includes all the attributes of the instance.

**Task 3**

1.Create a program by making available the functions of the Bird class that was defined earlier.

from Bird import \*

2. Add a statement to create an instance of the class.

zola = Bird ('Beep, beep!')

3. Then add a loop to display the values ​​of all the built-in attributes of the instance.

print ('\ nBuilt-in Instance Attributes ...')

for attrib in dir (zola):

if attrib [0] == '\_':

print (attrib)

4. Add a loop to list all elements from the class dictionary.

print ('\ nClass Dictionary ...')

for item in Bird .\_\_ dict\_\_:

print (item, ':', Bird .\_\_ dict \_\_ [item])

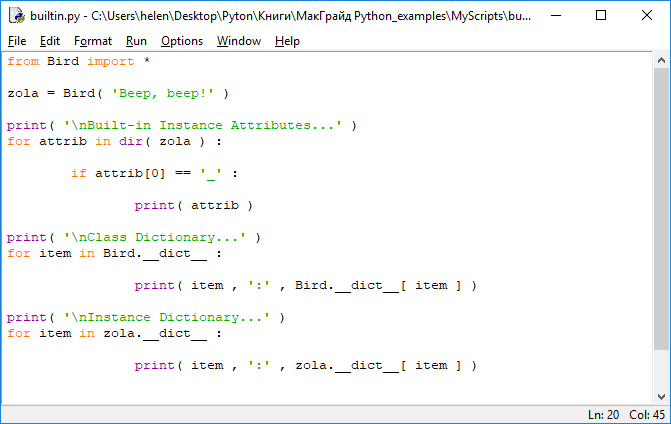
5. Add a loop to display all items from the instance dictionary.

print ('\ nInstance Dictionary ...')

for item in zola .\_\_ dict\_\_:

print (item, ':', zola .\_\_ dict \_\_ [item])

6. Save the file - you will see a list of built-in attributes





This program instantiates the class first, so the **\_\_init \_\_ ()** method increments the count until the instance dictionary is output

**5. Garbage collection (Сборка мусора)**

When an object is created - an instance of a class, a unique memory address is allocated for it. The unique address in memory can be viewed using the built-in function **id ()**

Python automatically performs garbage collection to free up memory, periodically deleting unnecessary objects, such as class instances, as a result of which their addresses in memory become free.

To destroy (для уничтожения) an instance of a class, you can also use the destructor - the **\_\_del \_\_ ()** method, which explicitly frees up the occupied memory space.

**Task 4**

1. Create a Python program, Songbird.py, by declaring a class with an initialization method in which two instance variables are created and outputting the value of one of those variables.

class Songbird:

def \_\_init \_\_ (self, name, song):

self.name = name

self.song = song

print (self.name, 'Is Born ...')

2. Add a method that simply displays the values of both variables.

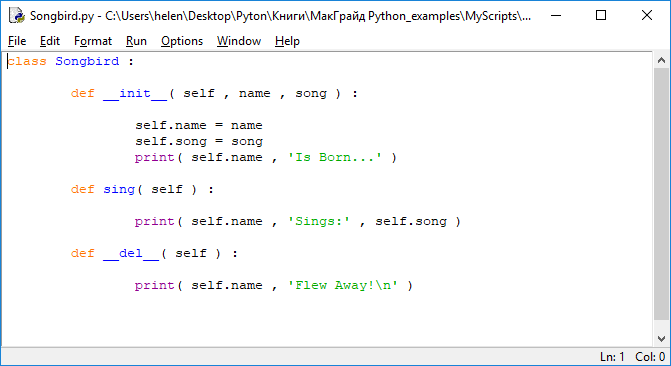
def sing (self):

print (self.name, 'Sings:', self.song)

3. Add a destructor method to confirm the destruction of the class instance, then save the file.

def \_\_del \_\_ (self):

print (self.name, 'Flew Away! \ n')



4. Create a second program by importing the functions of the Songbird class.

from Songbird import \*

5. Create an instance of the class, and then print the values ​​of its attributes and its unique identifier.

bird\_1 = Songbird ('Koko', 'Tweet, tweet! \ n')

print (bird\_1.name, 'ID:', id (bird\_1))

bird\_1.sing ()

6. Delete the instance by calling the destructor method.

del bird\_1

7. Create two more instances of the class, and then output the values ​​of the instance attributes and their unique identifiers again.

bird\_2 = Songbird ('Louie', 'Chirp, chirp! \ n')

print (bird\_2.name, 'ID:', id (bird\_2))

bird\_2.sing ()

bird\_3 = Songbird ('Misty', 'Squawk, squawk! \ n')

print (bird\_3.name, 'ID:', id (bird\_3))

bird\_3.sing ()

8. Delete these instances by calling their destructors.

del bird\_2

del bird\_3

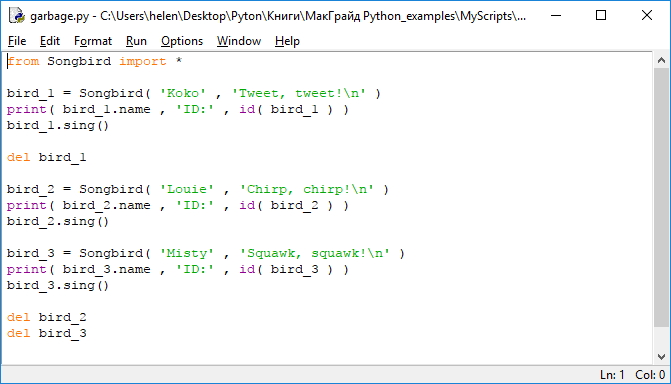
9. Save the garbage.py file and run the program - you will see the result of the garbage collector, which frees up computer memory.

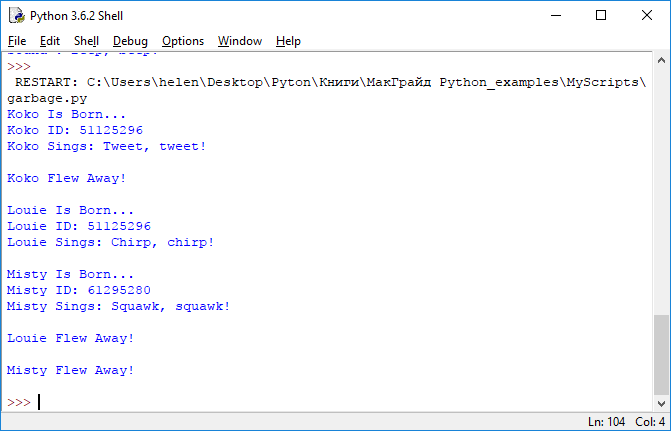
Songbird class - Koko instance

Songbird class - Louie instance

Songbird class - Misty instance

Note that the second instance created in this example is allocated a memory location that was freed after the first instance was deleted.





**6. Inheritance of properties (Наследование свойств)**

In Python, a class can be created either as brand new (новый) or derived (производный) from the existing one.

It is important to note that the derived class inherits the members of the parent (base) class from which it derives, in addition to its own members.

The ability to inherit members from the base class allows *you to create derived classes that have some common properties* that were defined for the base class.

**Task 5**

1. Create a program Polygon.py that declares a base class with two class variables and a method that sets the values for these variables.

class Polygon:

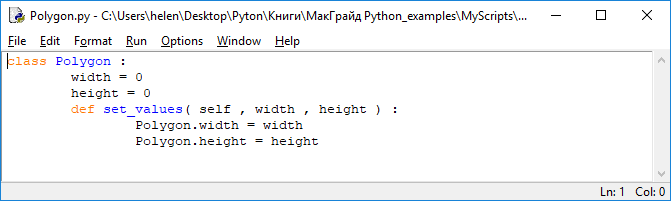
width = 0

height = 0

def set\_values (self, width, height):

Polygon.width = width

Polygon.height = height



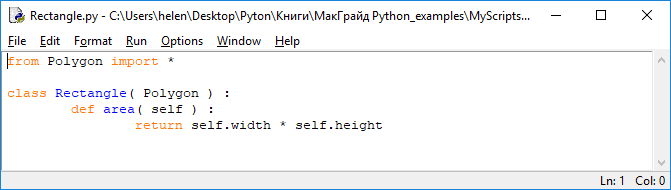
2. Next, create a second program, Rectangle.py, with a derived class definition that contains a method that returns the value of the class variables you want.

from Polygon import \*

class Rectangle (Polygon):

def area (self):

return self.width \* self.height



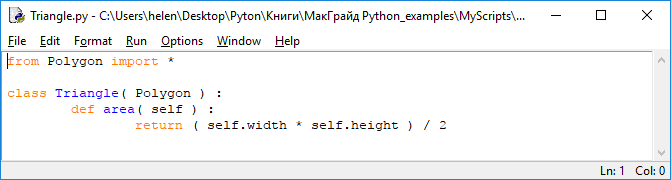
3. Now create a third program, Triangle.py, that defines a derived class and a method that returns the value of the class variables.

from Polygon import \*

class Triangle (Polygon):

def area (self):

return (self.width \* self.height) / 2



4. Save the three class files, then create a new program inherit.py, making the functions of both derived classes available.

from Rectangle import \*

from Triangle import \*

5. Create each of the derived classes.

rect = Rectangle ()

trey = Triangle ()

6. Call the class method inherited from the base one, passing the arguments for assigning values ​​to the class variables.

rect.set\_values ​​(4, 5)

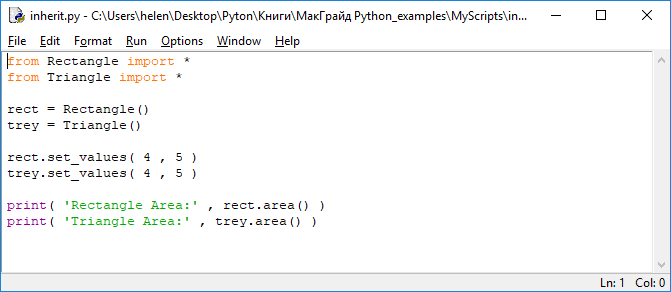
trey.set\_values ​​(4, 5)

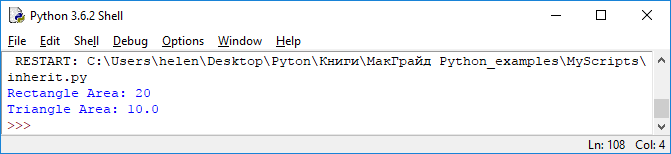
7. Finally, print the result of working with variables inherited from the base class.

print ('Rectangle Area:', rect.area ())

print ('Triangle Area:', trey.area ())

8. Save the inherit.py file and run the program - you will see the result of using inherited properties.





**7. Redefining (переопределение) Basic Methods**

If you declare a method in the derived class that has the same name and the same number of arguments as any method in the parent class, then this method is «***Redefined»*** (переопределяется)

In this case, the method from the base class becomes as if hidden and inaccessible until it is explicitly called using the name of the base class.

If a default argument is defined in the base method, then it will be used when explicitly calling the base class method, and for those arguments for which no default values ​​are specified, arguments from the overridden methods will be substituted.

**Task 6**

1. Create a Person.py program that declares a base class and method to initialize an instance variable, and a second method to display the value of that variable.

class Person:

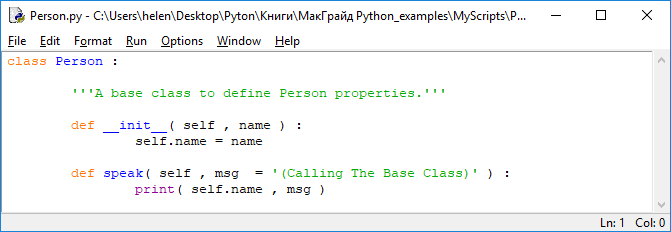
'' 'Base class.' ''

def \_\_init \_\_ (self, name):

self.name = name

def speak (self, msg = '(Calling The Base Class)'):

print (self.name, msg)



**2**. Create a program Man.py that declares a derived class with a method that overrides the second method of the base class.

from Person import \*

'' 'Derived class.' ''

class Man (Person):

def speak (self, msg):

print (self.name, ': \ n \ tHello!', msg)

3. Create a program Hombre.py that also declares a derived class and contains a method that again overrides the same method from the base class.

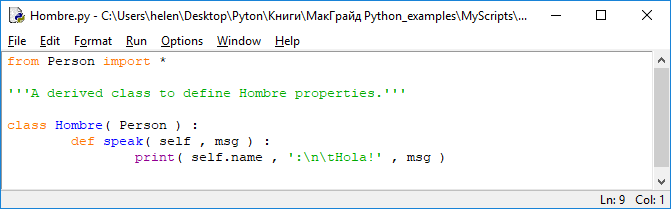
from Person import \*

'' 'Derived class.' ''

class Hombre (Person):

def speak (self, msg):

print (self.name, ': \ n \ tHola!', msg)



4. Save the three class files, then start a new override.py program making the properties of both derived classes available.

from Man import \*

from Hombre import \*

5. Instantiate each of the derived classes by initializing the instance variable name.

guy\_1 = Man ('Richard')

guy\_2 = Hombre ('Ricardo')

6. Now call the overridden methods of each of the derived classes, assigning different values ​​to the msg argument.

guy\_1.speak ('It \' s a beautiful evening. \ n ')

guy\_2.speak ('Es una tarde hermosa. \ n')

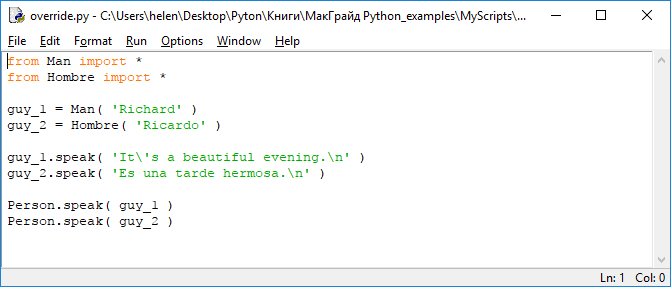
7. Call the method of the base class explicitly, passing a reference to each of the derived classes, but not passing a value for the msg argument, leaving the default value.

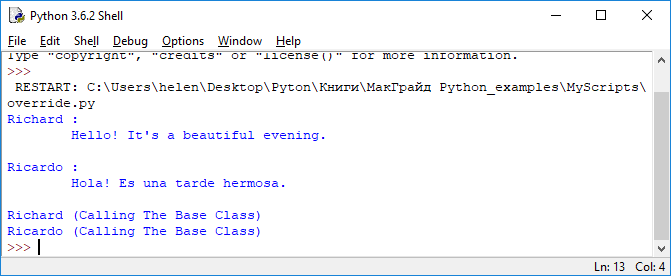
Person.speak (guy\_1)

Person.speak (guy\_2)

8. Save the override.py file and run the program - you will see the results of overriding the base class methods.

For methods to be overridden, their declarations must match exactly.





**8. Implementation of polymorphism (Реализация полиморфизма)**

The term polymorphism (from Greek, meaning "many forms") describes *the ability to assign an element to different semantic meanings depending on the context in which it is used.*

*for example*

In Python, the + character can be described as polymorphic, because it represents:

   - or arithmetic addition operator in the context of numerical operands

   - or a string concatenation operator, if the operands are not numbers, but characters

Class methods in Python can also be polymorphic, because the Python language uses the so-called *duck typing (утиная типизация)*- the meaning of which is that ...

"If she walks like a duck, swims like a duck, and quacks like a duck, then this bird can be considered a duck"

«если она ходит, как утка, плавает, как утка, и крякает, как утка, то эту птицу можно считать уткой»

In a language with duck typing, you can create a function in which an object of any type is taken, and the methods of this object are called.

If the object actually has callable methods (it can be considered a duck), then they are executed, otherwise the function signals an execution error.

Methods of the same name can be created for several classes, in which case each created instance of these classes will execute the corresponding version of the method.

**Task 7**

1. Create a **Duck.py** program by declaring a class with two methods that output strings that are unique to it.

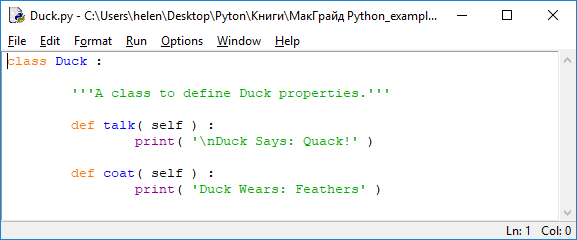
class Duck:

def talk (self):

print ('\ nDuck Says: Quack!')

def coat (self):

print ('Duck Wears: Feathers')



2. Create a program **Mouse.py**, declaring another class with two methods with the same names as the first, displaying their own, unique for this class, string values.

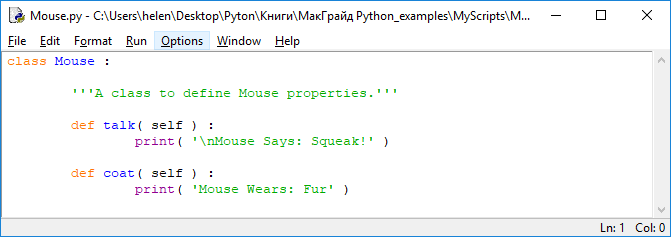
class Mouse:

def talk (self):

print ('\ nMouse Says: Squeak!')

def coat (self):

print ('Mouse Wears: Fur')



3. Save both class files, then create a program polymorph.py, making the functions of both classes available in it.

from Duck import \*

from Mouse import \*

4. Now define a function that takes any single object as an argument and tries to invoke methods on that object.

def describe (object):

object.talk ()

object.coat ()

5. Create instances of each of the two classes.

donald = Duck ()

mickey = Mouse ()

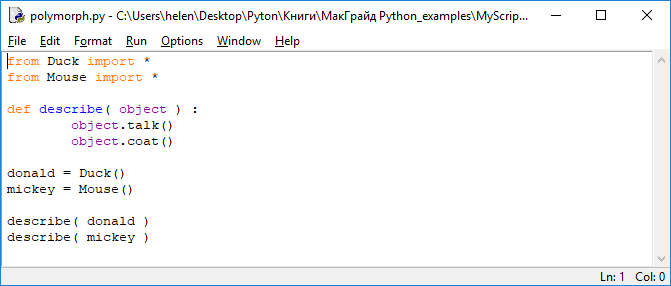
6. Add instructions that call the function, passing it first an instance of the first class, then the second.

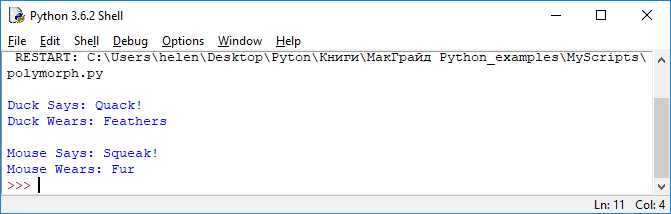
describe (donald)

describe (mickey)

7. Save the file in your working directory, open a command prompt and run the program - you will see how the methods associated with the corresponding classes are called.

A class can only contain one method with a specific name - method overloading in Python is not supported





**Conclusion**

• A class is a prototype of a data structure that describes the properties of objects using methods and attributes.

• Each class declaration begins with the keyword class. It is followed by an indented block of code that can contain the class docstring, class variables, and class methods.

• Class variables are globally scoped, while instance variables (declared inside method definitions) are only local.

• Instance variables are initialized when the class is instantiated and securely encapsulate the data in the class structure.

• Methods and attributes within a class can be accessed using the dot notation with the self prefix.

***The Basic literature***

1. Мэтиз Эрик.Изучаем Python. Программирование игр, визуализация данных, веб-приложения. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Библиотека программиста»).

2. [Mark Lutz](http://www.allitebooks.org/author/mark-lutz/). Learning Python. – O’Reilly, 2013.

3. Allen B. Downey. Think Python. - O’Reilly, 2015.

4. Brett Slatkin.  Effective Python: 59 Specific Ways to Write Better Python. - 2015. – 256 p.

5. [Mark Lutz. Programming Python: Powerful Object-Oriented Programming](https://www.amazon.com/Programming-Python-Powerful-Object-Oriented/dp/0596158106/ref=dp_ob_title_bk) O'Reilly Media, 2016. – 1552 p.

6. МакГрат Майк. Программирование на Python для начинающих. – М.: Эксмо, 2015. -  192 с.

***Additional***

7. Саммерфилд М. Программирование на Python 3. Подробное руководство. – Пер. с англ. – СПб.: Символ\_Плюс, 2009. – 608 с., ил.

8. Златопольский Д. М. Основы программирования на языке Python. – М.: ДМК Пресс, 2017. – 284 с.: ил.

***Internet resources***

http://www.python.org/

http://pythontutor.ru/

http://www.enthought.com/products/epdlibraries.php

http://www.edu.ru/modules.php?op=modload&name=Web\_Links&file=index&l\_op=viewlink

https://docs.python.org/3/tutorial/index.html

http://www.python.org/

http://pythontutor.ru/